CS 5800: Algorithms SEC 05 Summer Full 2024 (Seattle)
Synthesis1

Payal Chavan

06/07/2024

Chapter 1

Divide and Conquer Multiplication

1.1 Introduction

Divide and Conquer is a strategy that involves breaking a larger problem into subproblems of the same problem type, and recursively solving the subproblems independently. The solutions of these subproblems are then combined to get the solution for the larger problem.
The key steps involved in Divide and Conquer Algorithm are:

1. **Divide:** Break the larger problem into smaller manageable subproblems.
2. **Conquer:** Recursively solve each of the subproblems.
3. **Merge:** Combine the solutions of subproblems to obtain the final solution for the main problem.

1.2 Applications of Divide-and-Conquer

Some of the practical applications of Divide and Conquer Algorithm are:

1. Binary Search.
2. Quick Sort.
3. Merge Sort.
4. Closest Pair of Points.
5. Cooley-Tukey Fast Fourier Transform (FFT) Algorithm.
6. Strassen's Matrix Multiplication.
7. Karatsuba Algorithm for Fast Multiplication.

Now, let us see how to implement Divide and Conquer Algorithm to multiply two n-bit integers using "Karatsuba Algorithm for Fast Multiplication". This technique is more efficient than the one that we learnt in our elementary school!

1.3 Integer Multiplication

Globally, computers multiply massive amounts of data every day. Even while the majority of computers only need a few microseconds to complete an activity, the variations in processing times can add up over time. Therefore, the general question of the maximum speed at which two n-bit values can be multiplied has both theoretical and practical significance.

Carl Gauss introduced the multiplication of 2 complex numbers. $(a + bi)(c + di) = [ac - bd] + [ad + bc]i$.
To obtain the solution, it requires 4 multiplications and 2 additions. This is computationally expensive in the real world.

Gauss then optimized the 4 multiplications to 3 multiplications by- $(ad + bc) = (a + b)(c + d) - ac - bd$.
In terms of running time, this would still be insignificant approach when applied recursively!

Karatsuba improved Gauss's algorithm by speeding up the integer multiplication. The basic idea is to take a given multiplication problem and break it down into smaller subproblems. Then, you solve the subproblems recursively and combine the results to get the answer to the bigger problem.

Using Gauss optimization, Karatsuba's innovative approach was to replace some multiplications with additional adds in this divide-conquer-and-glue strategy. Karatsuba's approach is quite effective for huge numbers, either in binary or decimal. Our computers may be performing this kind of work in the background to respond to us in a split second quicker.

To solve integer multiplication of two n-bit integers (X and Y) using divide-and-conquer technique is given by the formula:

$$X \times Y = (2^{n/2}X_L + X_R)(2^{n/2}Y_L + Y_R)$$
$$\implies 2^n X_L Y_L + 2^{n/2}(X_L Y_R + X_R Y_L) + X_R Y_R$$

where, X is split into 2 halves (X_L = left half, X_R = right half), and Y is split into 2 halves (Y_L = left half, Y_R = right half).

If we split X and Y of n-bits into two halves, we get (n/2) bits of subproblems. There are 4 significant multiplications $(X_L Y_L, X_L Y_R, X_R Y_L, X_R Y_R)$, which would make 4 recursive calls. This algorithm can be solved by dividing a problem to get 4 subproblems of half the size (n/2), recursively solving each subproblem, and then combining the solutions in linear time.

Hence, $T(n) = 4T(n/2) + O(n)$, and the time complexity is $\mathcal{O}(n^2)$.

This running time of $\mathcal{O}(n^2)$ is similar to the running time of our elementary school multiplications technique. So the efficiency of the algorithm hasn't yet improved.

1.4 Pseudocode Algorithm

Now let's see the pseudocode algorithm:

```
function binaryMultiply (X, Y)
Input: n-bit positive integers X and Y
Output: Their product

if $n = 1$: return $X * Y$

$X_L, X_R$ = leftmost[n/2], rightmost[n/2] bits of X
$Y_L, Y_R$ = leftmost[n/2], rightmost[n/2] bits of Y

$P1$ = multiply$(X_L, Y_L)$
$P2$ = multiply$(X_R, Y_R)$
$P3$ = multiply$(X_L + X_R, Y_L + Y_R)$

return $P1 \times 2^n + (P3 - P1 - P2) \times 2^{n/2} + P2$
```

This algorithm has an improvement in running time of $\mathcal{O}(n^{1.59})$, given by -
$T(n) = 3T(n/2) + \mathcal{O}(n)$.

1.5 Advantages of Divide-and-Conquer Multiplication

The Karatsuba algorithm has several advantages for binary multiplication:

- Efficiency: The Karatsuba algorithm has a time complexity of $\mathcal{O}(n^{\log_2(3)}) = \mathcal{O}(n^{1.59})$, which is faster than the traditional multiplication algorithm with a time complexity of $\mathcal{O}(n^2)$. This makes it more efficient for large numbers.
- Divide and Conquer: The algorithm uses a divide and conquer approach, breaking the problem down into smaller, easier-to-solve problems. This makes the algorithm easier to understand and implement.
- Less Multiplication Operations: The algorithm reduces the number of basic multiplication operations required. This can lead to significant time savings, especially for large numbers.
- Scalability: The algorithm scales well with the size of the input. As the number of digits in the input increases, the Karatsuba algorithm becomes increasingly more efficient compared to traditional multiplication.
- Recursive: The algorithm is recursive, which can make the code more compact and easier to debug and maintain.

Now let's take an example to better understand how the binary multiplication works using divide-and-conquer approach.

1.6 Exercise Problem

Question: Use the divide-and-conquer integer multiplication algorithm to multiply the two binary integers given below:

$X = 1010, Y = 1001$

Solution:

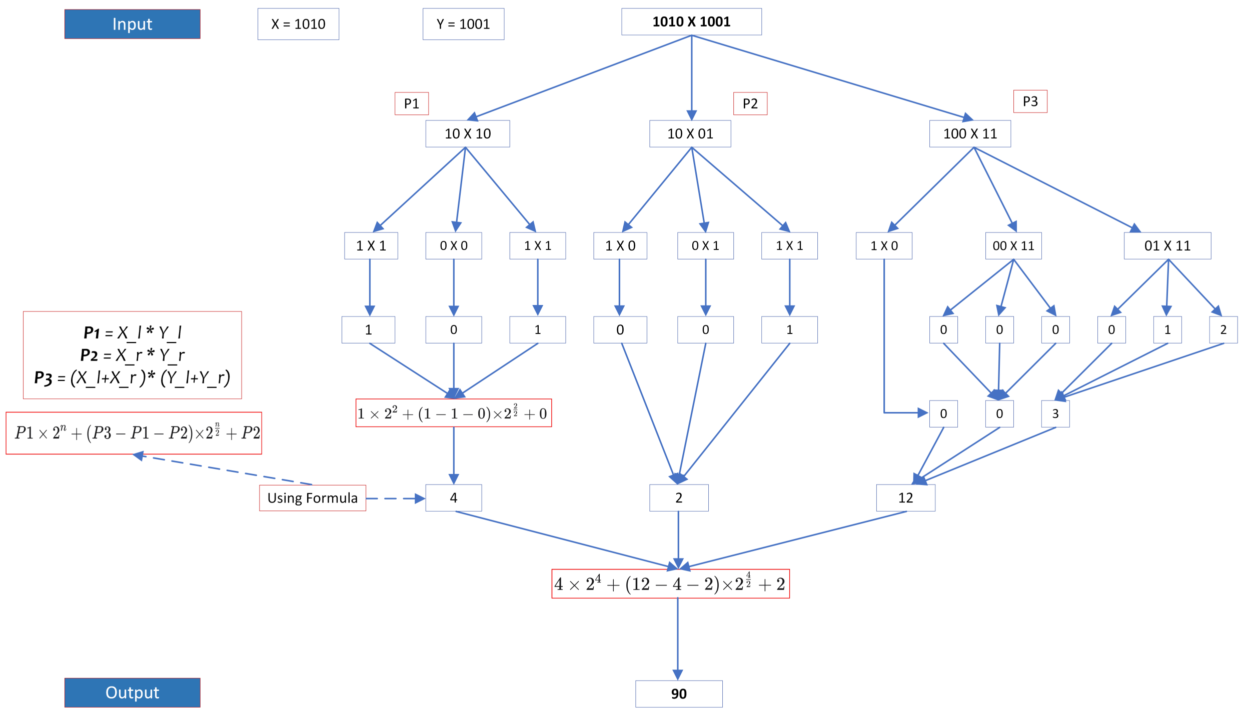
Input
X = 1010
Y = 1001
1010 X 1001
P1
P2
P3
10 X 10
10 X 01
100 X 11
1 X 1
0 X 0
1 X 1
1 X 0
0 X 1
1 X 1
1 X 0
00 X 11
01 X 11
1
0
1
0
0
1
0
0
0
0
1
2
P1 = X_l * Y_l
P2 = X_r * Y_r
P3 = (X_l+X_r)* (Y_l+Y_r)
$P1 \times 2^n + (P3 - P1 - P2) \times 2^{\frac{n}{2}} + P2$
$1 \times 2^2 + (1 - 1 - 0) \times 2^{\frac{2}{2}} + 0$
0
0
3
Using Formula
4
2
12
$4 \times 2^4 + (12 - 4 - 2) \times 2^{\frac{4}{2}} + 2$
Output
90

Figure 1: Binary Multiplication using Divide-and-Conquer.

Call 1

Step 1:

$X = 1010, Y = 1001$

$X_L = 10$

$X_R = 10$
$Y_L = 10$
$Y_R = 01$

Step 2:
$P_1 = X_L * Y_L =$
$\implies 10 * 10$

$P_2 = X_R * Y_R =$
$\implies 10 * 01$

$P_3 = (X_L + X_R) * (Y_L + Y_R)$
$\implies (10 + 10) * (10 + 01)$
$\implies 100 * 11$

NOTE Here we are only showing recursive call for $P_1 = binary - multiplication(X_L, Y_L)$. Because this recursive call will be similar for P_2 and P_3.

Next Iteration: ***Call 2***

$X = 10, Y = 10$

$X_L = 1$
$X_R = 0$
$Y_L = 1$
$Y_R = 0$

$P_1 = X_L * Y_L =$
$\implies 1 * 1$

$P_2 = X_R * Y_R =$
$\implies 0 * 0$

$P_3 = (X_L + X_R) * (Y_L + Y_R)$
$\implies (1 + 0) * (1 + 0)$
$\implies 1 * 1$

Next Iteration: ***Call 3***

$X = 1, Y = 1$
Since, this is our base condition
return $X * Y \implies 1 * 1 \implies 1$

Coming back from recursive call, we are at ***Call 2***
we have all values for P1, P2 and P3
$P1 = 1$
$P2 = 0 \rightarrow$ *similar callback result from recursive binary-multiplication*
$P3 = 1 \rightarrow$ *similar callback result from recursive binary-multiplication*

Using formula $P1 \times 2^n + (P3 - P1 - P2) \times 2^{n/2} + P2$
$\implies 1 * 2^2 + (1 - 1 - 0) * 2^{2/2} + 0$
$\implies 4 + 0 + 0$
$\implies 4$
we return 4 from this function call.

Coming back from recursive call, we are at ***Call 1***
$P1 = 4$
$P2 = 2 \rightarrow$ *similar callback result from recursive binary-multiplication*
$P3 = 12 \rightarrow$ *similar callback result from recursive binary-multiplication*

Using the formula mentioned above
Calculate multiplication
$\implies 4 * 2^4 + (12 - 4 - 2) * 2^{4/2} + 2$
$\implies 4 * 2^4 + (12 - 4 - 2) * 2^{4/2} + 2$
$\implies 64 + 6 * (2^{4/2}) + 2$
$\implies 64 + 6 * 4 + 2$
$\implies 64 + 24 + 2$
$\implies 90$

We have our answer from the function
$90 = (10 \times 9)$ decimal
$90_{10} = (1010 \times 1001) = X \times Y$ binary

1.7 Coding Problem

Question: Compute binary multiplications for the following 5 test cases using Divide and Conquer Approach:

1. Test case 1: $(1100, 1010)$
2. Test case 2: $(101, 101)$
3. Test case 3: $(111, 111)$
4. Test case 4: $(100, 100)$
5. Test case 5: $(1111, 1111)$

Solution: Please refer the attached Binary-Multiplication Python file for the solution.

Chapter 2
Mergesort

2.1 Introduction

Mergesort is a type of sorting algorithm that uses a divide-and-conquer approach. It sorts the input array by recursively splitting it into smaller subarrays, sorting those subarrays, and merging them back together to obtain the resulting array.

The key steps involved in Mergesort Algorithm are:

1. **Divide:** Divide the original array into two equal halves, until it can no more be divided.
2. **Conquer:** Recursively sort each of the sub-arrays using mergesort algorithm.
3. **Merge:** The sorted sub-arrays are then merged back together to obtain the resulting sorted array.

2.2 Applications of Mergesort

Some of the practical applications of Mergesort Algorithm are:

1. Inversion count problem.
2. Sorting large datasets.
3. External sorting.
4. Finding the median of an array.
5. E-commerce applications.

2.3 Pseudocode Algorithm

The pseudocode algorithm for recursive-mergesort is given by:

NOTE Here we are considering indexing starting from 1 for array.

```
function mergesort ($a[1 \cdots n]$)
Input: An array of numbers $a[1 \cdots n]$
Output: A sorted version of this array

if $n > 1$:
    return merge(mergesort ($a[1 \cdots [n/2]]$)),
    mergesort ($a[n/2] + 1 \cdots n$)

else:
    return a
```

```
function merge ($x[1\cdots k], y[1\cdots l]$)

if x is empty : return y
if y is empty : return x
if $x[1] \leq y[1]$:
    return $x[1] \circ merge(x[2\cdots k], y[1\cdots l])$
else:
    return $y[1] \circ merge(x[1\cdots k], y[2\cdots l])$
```

The mergesort function is used to recursively divide the array into smaller sub-arrays. This process continues until each sub-array contains only one element(i.e., base case).

The merge function is used to combine the sorted sub-arrays into a final sorted array.

The key step here is merging where we compare the elements from both the right and left sub-arrays to place them in the correct order.

2.4 Time and Space Complexity Analysis

Mergesort is a recursive algorithm and it's time and space complexity can be analyzed by the following steps $\longrightarrow$

1. Consider T(n) as the time complexity for sorting 'n' elements using mergesort algorithm.
   To calculate T(n), let's break down the time complexities as follows
   (a) **Divide part:** The time complexity to divide the array into two halves is $\mathcal{O}(1)$.
   (b) **Conquer part:** We are recursively solving two sub-problems, each of size n/2. Hence, the overall time complexity of conquer part is $2T(n/2)$.
   (c) **Merge part:** The time complexity for combining two sorted arrays having 'n' elements is $\mathcal{O}(n)$.
2. To calculate, T(n), add the time complexities of divide, conquer, and merge part $\implies$
   $T(n) = \mathcal{O}(1) + 2T(n/2) + \mathcal{O}(n)$
   $\implies 2T(n/2) + \mathcal{O}(n)$
3. Based on the above recurrence relation $T(n) = 2T(n/2) + \mathcal{O}(n)$, we can find out the time complexity. When we draw the recursion tree, we get the recurrence tree height of $\log n$. The list of size 'n' is divided into a max of $\log n$ parts, and the merging of all sublists into a single list takes $\mathcal{O}(n)$ time. So the total operations would be $\mathcal{O}(n) * \mathcal{O}(\log n)$. Hence, the time complexity of mergesort algorithm is $\mathcal{O}(n \log n)$.
4. **Best Case:** When the array is sorted, we have time complexity of $\mathcal{O}(n \log n)$.
5. **Average Case:** When the array is randomly ordered, we have time complexity of $\mathcal{O}(n \log n)$.
6. **Worst Case:** When the array is sorted in reverse order, we have time complexity of $\mathcal{O}(n \log n)$.
7. The space complexity of mergesort algorithm is $\mathcal{O}(n)$. This is due to the additional space required by the temporary array during merging process.

2.5 Advantages and Disadvantages of Mergesort

2.5.1 Advantages:

1. Mergesort is a stable sorting algorithm i.e., it preserves the relative order of equal elements in the input array.
2. The divide-and-conquer approach of mergesort makes it easy to implement.
3. Since mergesort has a worst-case time complexity of $\mathcal{O}(n \log n)$, it's well-suited for handling large datasets.

2.5.2 Disadvantages:

1. During the sorting process, merge sort needs extra memory to store the merged sub-arrays.
2. In applications where memory usage is a concern, the fact that merge sort is not an in-place sorting algorithm can be a disadvantage because it requires additional memory to store the sorted data.
3. Due to its recursive nature and additional space requirements, mergesort may be less efficient for sorting very small arrays compared to some other algorithms like Insertion Sort or Bubble Sort.

Now let's take an example to better understand how the mergesort works.

2.6 Exercise Problem

Question: Given an array $[6, 3, 8, 9, 5, 4]$, perform Mergesort and also find it's time complexity.

Solution:

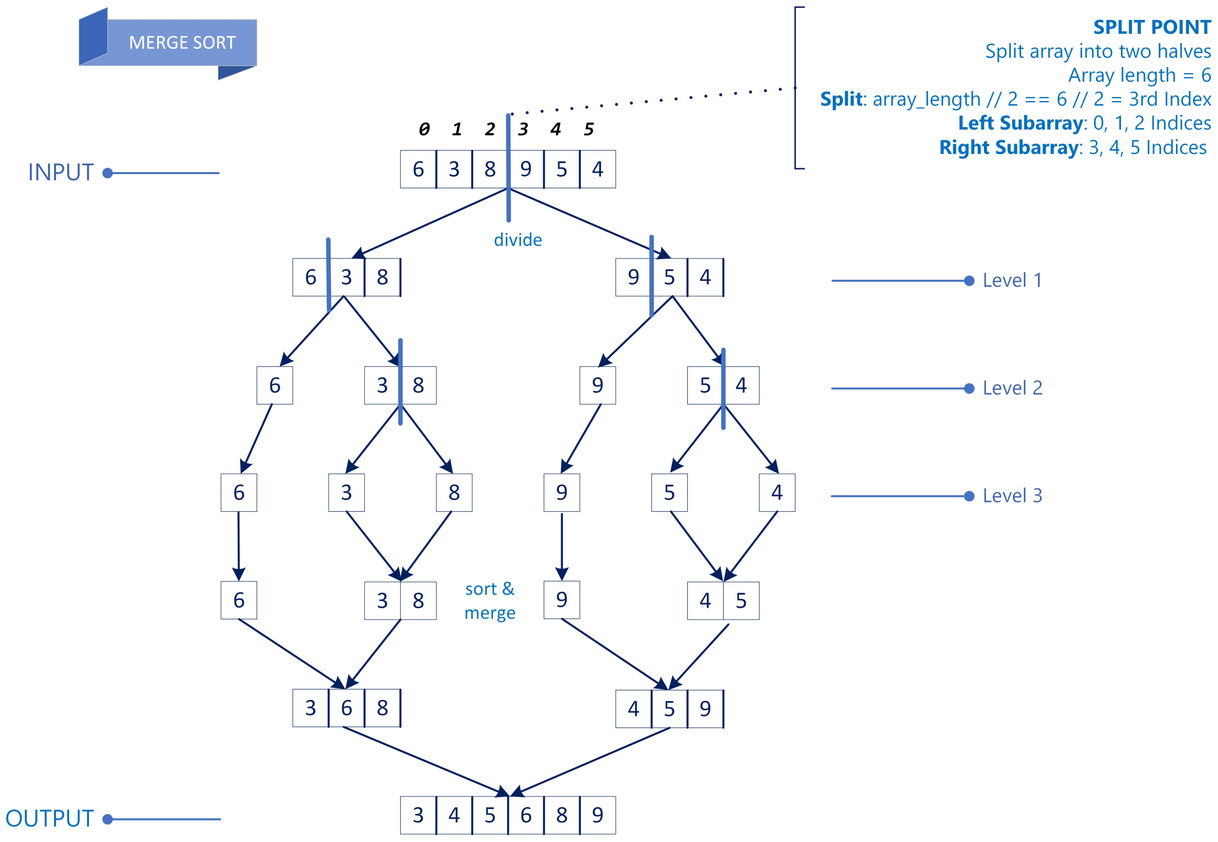

Figure 2: Mergesort Implementation

Let us see step-by-step on how to sort an array using Mergesort.
Given:- Input array: $[6, 3, 8, 9, 5, 4]$

Step 1: Divide the given array into two halves $\longrightarrow$

Note that, here we are dividing the array based on the split point (splits array into two halves).
We have, array length $= 6$
array length $= 6//2 = 3^{\text{rd}}$ index.

$[6, 3, 8]$ $[9, 5, 4]$

Step 2: Again divide the sub-array until it's left only with only one element. $\longrightarrow$
$[6]$ $[3, 8]$ $[9]$ $[5, 4]$

Step 3: Divide the sub-arrays further so that we are left with only single element.

$[6]$ $[3]$ $[8]$ $[9]$ $[5]$ $[4]$

Step 4: Now sort and merge the sub-arrays of both left half and right half.

$[6]$ $[3, 8]$ $[9]$ $[4, 5]$

Step 5: Repeat the above step to get back the sorted left and right sub-arrrays.

$[3, 6, 8]$ $[4, 5, 9]$

Step 6: Once again perform final sort and merge step to obtain the resulting sorted array.

Hence, we get the final output array as $\longrightarrow [3, 4, 5, 6, 8, 9]$

Now, let's find the time-complexity of the given input array.

We will use Master Theorem to find the time-complextity of Mergesort Algorithm,

The Master Theorem is given as follows:
If $T(n) = aT([n/b]) + \mathcal{O}(n^d)$ for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$\begin{cases} T(n) = \mathcal{O}(n^d) & \text{if } d > \log_b a \\ T(n) = \mathcal{O}(n^d \log n) & \text{if } d = \log_b a \\ T(n) = \mathcal{O}(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Here, a $\rightarrow$ no. of subproblems,
b $\rightarrow$ the factor by which the problem size is reduced,
f(n) $\rightarrow$ cost of dividing the problem and combining the results.

For Mergesort Algorithm, we have derived T(n) in section 2.4.
$T(n) = 2T(n/2) + \mathcal{O}(n)$

Comparing the given recurrence relation with the Master Theorem, we have:
$a = 2, b = 2, n = 6, d = 1$

Calculating, $\log_b a = \log_2 2 = 1$

Since, $d = \log_b a$ i.e., $1 = 1$
This satisfies case 2 of the Master Theorem.

$T(n) = \mathcal{O}(n^d \log n) = \mathcal{O}(6^1 \log 6) = \mathcal{O}(1)$

Therefore, the time complexity of the given input array using mergesort algorithm is $\mathcal{O}(6 \log 6)$ i.e., $\mathcal{O}(1)$.

2.7 Coding Problem

Question:
Implement the given test cases of arrays using Mergesort Algorithm.

1. Test Case 1: $[5, 1, 1, 2, 0, 0]$
2. Test Case 2: $[5, 2, 3, 1]$
3. Test Case 3: $[9, 8, 7, 6, 5, 4, 3]$

Solution:
Please refer the attached MergeSort python file for the solution.

Chapter 3
Quicksort

3.1 Introduction

Quicksort is a sorting algorithm that follows the Divide and Conquer approach. It selects an element as a pivot and rearranges the array so that elements smaller than the pivot are on its left, and elements greater than the pivot are on its right. This process continues recursively until the entire array is sorted.

The key steps involved in Quicksort Algorithm are:

1. **Select Pivot:** Choose an element from the array as the pivot.

   There are different ways of choosing pivot element:

   (a) **First Element:** Always choose the first element as the pivot.

   (b) **Last Element:** Always choose the last element as the pivot.

   (c) **Random Element:** Select a random element as the pivot.

   (d) **Middle Element:** Use the middle element as the pivot.

2. **Partition Array:** Rearrange the elements so that smaller ones are on the left of the pivot, and greater ones are on the right.

3. **Recursively Sort:** Recursively sort each sub-array until it contains a single element (base case).

However, how is the "rearranging" of the elements around the pivot value truly accomplished? Let's look at a Quicksort Swapping to better understand this.

Understanding Swapping is a key idea in Quicksort that is necessary to fully comprehend its effectiveness. The fact that quicksort occupies little more space while sorting is one of the several reasons it is a favored algorithm. Quicksort sorts elements in-place rather than creating a duplicate array. Quicksort uses pointers or references to track swapping in order to sort data in-place quickly and effectively.

Direct sorting of the input data is done by an in-place algorithm. The drawback of an in-place algorithm is that it may change the data while it processes, thus wiping out the original values. Because an in-place algorithm uses very little memory space—typically a constant size of O(1)—it is space-efficient.

3.2 Applications of Quicksort

Some of the practical applications of Quicksort Algorithm are:

1. Information Searching.
2. Commercial Computing.
3. Numerical Analysis.
4. Operational Research.
5. Event-Driven Simulation.

3.3 Pseudocode Algorithm

The pseudocode algorithm for recursive-quicksort is given by:

NOTE Here we are considering indexing starting from 1 for array.

```
function Quicksort (array, start, end)
base case size $\leq 1$
if start $\geq$ end then
return
end if
pivot_index = Partition(array, start, end)
Quicksort(array, start, pivot_index $-1$ )
Quicksort(array, pivot_index $+1$, end)
end function
```

```
function Partition (array, start, end)
pivot_value = array[end]
pivot_index = start
loop index from start to end
if array[index] $\leq$ pivot_value
swap array[index] with array[pivot_index]
pivot_index = pivot_index $+1$
end if
end loop
return pivot_index $-1$
```

Let's walk through the pseudocode of the Quicksort function.

1. The base case checks if the array size is less than or equal to 1. If start is greater than or equal to end, it means the array is either empty or contains only one element. In such cases, the function returns immediately without any further sorting.
2. The Partition function is used to determine the pivot_index. The array's elements are rearranged by the Partition function so that any element that is greater than the pivot value is on the right and any element that is less than or equal to the pivot value is on the left. The pivot value's final position upon partitioning is indicated by the pivot_index.
3. The Quicksort function recursively calls the two sub-arrays $\rightarrow$
   left sub-array: start to $pivot\_index - 1$
   right sub-array: $pivot\_index + 1$ to end

Now let's have a look at the Partition function step-by-step:

1. In the Quicksort algorithm, the Partition algorithm is an essential stage. It guarantees that the chosen pivot element is positioned correctly within the array. Quicksort's efficiency is affected by the pivot and partitioning strategy choices.
2. The pivot_value is set to the last element of the given array. And the pivot_index is initialized to the start index.
3. The loop runs from the start to the end (inclusive) of the array. Every element at the current index is replaced with the element at the pivot_index if the element is less than or equal to the pivot_value. The pivot_index is incremented by 1. By rearranging the elements in this way, all elements that are less than or equal to the pivot value are positioned on the left side.

4. After the loop completes, the $pivot\_index - 1$ represents the final position of the pivot value. Every element to the left of this index is greater than or equal to the pivot value, while every element to the right of this index is smaller.

3.4 Time and Space Complexity Analysis

Analyzing the Quicksort algorithm may be somewhat challenging, particularly because the pivot value is chosen at random and has a significant impact on the algorithm's performance. Thus, we will discuss the time complexity of quicksort in terms of two scenarios: the worst-case scenario and the average-case scenario.

1. Consider T(n) as the time complexity for sorting 'n' elements using quicksort algorithm.
   To calculate T(n), let's break down the time complexities as follows:
   (a) **Divide part:** The time complexity to divide part is equal to the time complexity of the partition, which is $\mathcal{O}(n)$.
   (b) **Conquer part:** When solving a problem using quicksort, we recursively break it down into two sub-problems. The pivot choice during partitioning determines the size of these sub-problems. After partitioning, we have "k" elements in the left sub-array and "$n - k - 1$" elements in the right sub-array. The time complexity of the conquer part is the sum of the time complexities for sorting the left and right sub-arrays: $(T(k) + T(n - k - 1))$.
   (c) **Merge part:** The time complexity for combining two sorted arrays is $\mathcal{O}(1)$.
2. To calculate, T(n), add the time complexities of divide, conquer, and merge part $\implies$
   $T(n) = \mathcal{O}(n) + (T(k) + T(n - k - 1)) + \mathcal{O}(1)$
   $\implies T(k) + T(n - k - 1) + cn$
3. Now, let's solve this above recurrence relation to find out the time complexity of quicksort for worst-case scenario.
   (a) When the partition process consistently selects either the largest or smallest element as the pivot, the resulting partition becomes highly unbalanced. Specifically, one sub-array contains "$n - 1$" elements, while the other sub-array remains empty.
   (b) If we select the rightmost element as the pivot in quicksort, the worst-case scenario occurs when the array is already sorted in either increasing or decreasing order. In this situation, each recursive call results in an unbalanced partition.
   (c) For calculating time complexity, put $k = n - 1$ in the above formula of T(n).
   we get, $T(n) = T(n - 1) + T(0) + cn$
   $\implies \quad T(n) = T(n - 1) + cn$

   Expanding this above recurrence relation,

   $T(n) = T(n - 1) + cn$
   $= T(n - 2) + c(n - 1) + cn$
   $= T(n - 3) + c(n - 2) + c(n - 1) + cn$
   $\vdots$
   Summing up these equations, we get the arithmetic series,

   $\implies c + 2c + 3c + \cdots + c(n - 2) + c(n - 1) + cn$

   $\implies c(1 + 2 + 3 + \cdots + n - 1 + n - 2 + n)$

   $\implies c(n(n + 1)/2)$

$\implies \mathcal{O}(n^2)$

Therefore, the worst-case time complexity of quicksort is $\mathcal{O}(n^2)$.

4. Now, let's calculate the time complexity of quicksort for best-case scenario.
   (a) The best-case scenario arises when the partition process consistently selects the median element as the pivot. In this situation, we achieve a balanced partition, with both sub-arrays having a size of (n/2) each.
   (b) Let us assume that every recursive call results in the balanced partitioning situation.
   For calculating time complexity, put $k = n/2$ in the above formula of T(n).

   we have, $T(n) = T(k) + T(n - k - 1) + cn$

   $\implies T(n) = T(n/2) + T(n - n/2 - 1) + cn$

   $\implies T(n/2) + T(n/2 - 1) + cn$

   $\implies 2T(n/2) + cn$ (approx)

   Note that, this is similar to the recurrence relation of mergesort. And the solution to this was $\mathcal{O}(n \log n)$. We have proved this in chapter 2 (Mergesort).
   Therefore, the best-case time complexity of quicksort is $\mathcal{O}(n \log n)$.
5. Now, let's see the time complexity of quicksort for average-case scenario.
   (a) When we run the partition process of quicksort using random pivot value, we either have balanced or unbalanced splits. These good or bad splits will be distributed randomly across the recursion tree.
   (b) Hence, the cost of partition will be same for both good split and bad split, which is $\mathcal{O}(n)$. As a result, the height of the recursion will be $\mathcal{O}(\log n)$. Therefore, the average-case time complexity of quicksort is $\mathcal{O}(n \log n)$. (Similar to the best-case scenario).
6. **Worst-Case:** The worst-case space complexity for quicksort is $\mathcal{O}(n)$. This occurs when unbalanced partitioning results in a skewed recursion tree, requiring a call stack of size $\mathcal{O}(n)$.
7. **Best-Case:** The best-case space complexity for quicksort is $\mathcal{O}(\log n)$. This occurs due to the balanced partitioning, resulting in a balanced recursion tree with a call stack of size $\mathcal{O}(\log n)$.

3.5 Advantages and Disadvantages of Mergesort

3.5.1 Advantages:

1. It uses divide-and-conquer approach which makes it easier to implement.
2. It performs efficiently on large datasets.
3. It uses in-place sorting algorithm where we use memory only for recursion call stack, but not for manipulating the input.
4. Quicksort partitioning process is efficiently used to find the kth-smallest or kth-largest element in an unsorted array.

3.5.2 Disadvantages:

1. It poorly performs on smaller datasets.
2. Quicksort is an unstable sorting algorithm, which means it does not preserve the relative order of equal elements in the input array.
3. The worst-case time complexity of quicksort is $\mathcal{O}(n^2)$, which happens when the pivot is not chosen properly.

Now let's take an example to better understand how the quicksort works.

3.6 Exercise Problem

Question: Sort the given array $[8, 7, 6, 1, 0, 9, 2]$ using Quicksort Algorithm.

Solution:

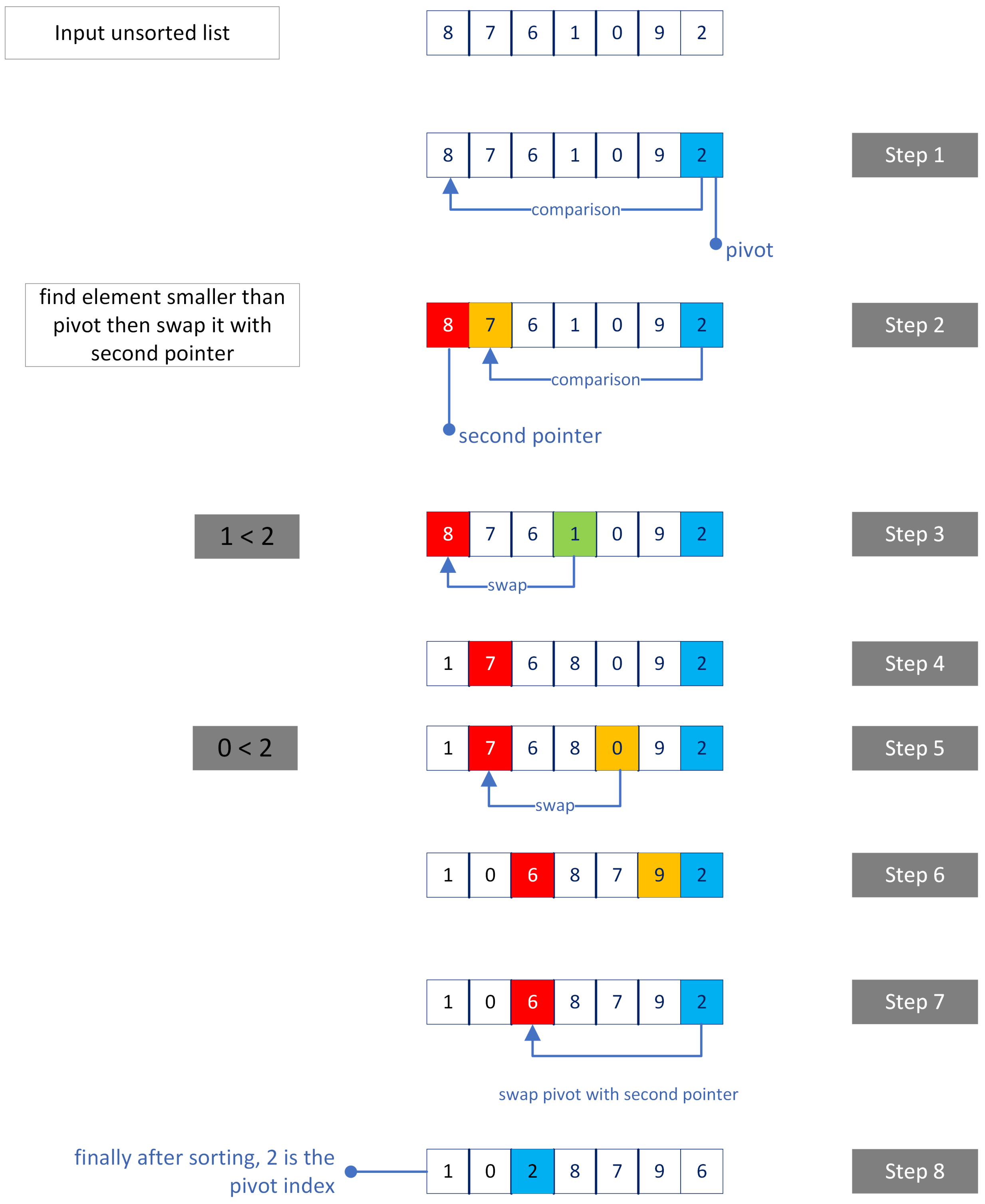

Figure 3: Quicksort Implementation Steps

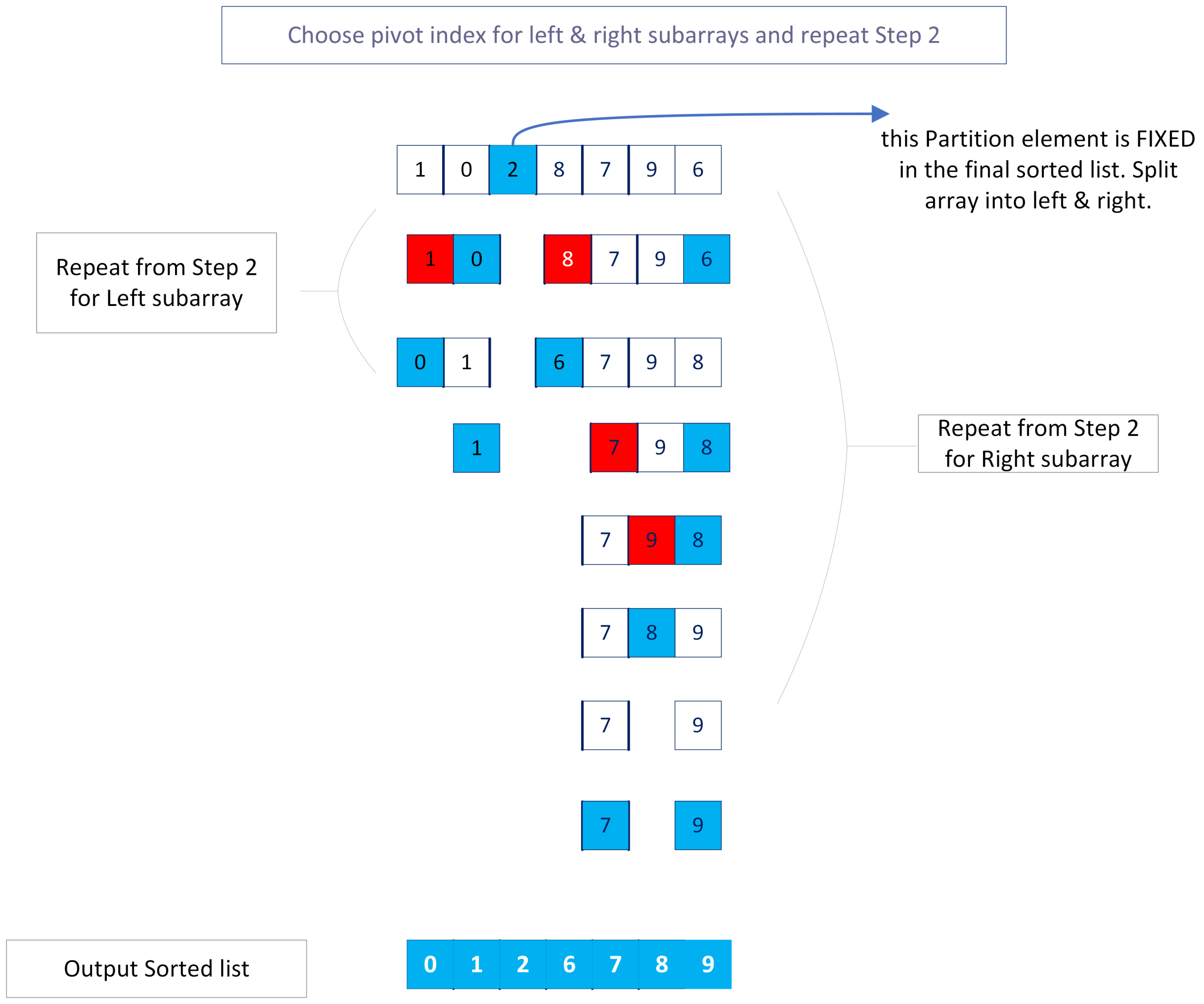
Choose pivot index for left & right subarrays and repeat Step 2
this Partition element is FIXED in the final sorted list. Split array into left & right.
Repeat from Step 2 for Left subarray
Repeat from Step 2 for Right subarray
Output Sorted list
1 0 2 8 7 9 6
1 0
8 7 9 6
0 1
6 7 9 8
1
7 9 8
7 9 8
7 8 9
7 9
7 9
0 1 2 6 7 8 9

Figure 4: Quicksort Implementation Steps

Let us see step-by-step on how to sort an array using Quicksort.
Given:- Input array:$[8, 7, 6, 1, 0, 9, 2]$

Step 1: Choose Pivot Let's choose 2 as the pivot element.(last element)

Step 2: Partition Array Partition the given array such that the elements greater than the pivot are at its right side, and elements smaller than the pivot are at its left side.

Now, let's start rearranging the elements accordingly.
elements less than $2 \longrightarrow [1, 0]$
elements greater than $2 \longrightarrow [8, 7, 9, 6]$

Now, we have two sub-arrays:
$[1, 0], 2, [8, 7, 9, 6]$

Step 3: Recursively sort sub-arrays We will apply this step to recursively sort each sub-array, until we reach the base case, i.e., it should contain only a single element in the sub-array.

For sorting the left sub-array:

Choose the pivot element: choosing 0 (last element).

Since, we only have 2 elements in the list. 0 is the pivot and by following above process left sub-array is sorted with $[0, 1]$
$0, 1, 2, [8, 7, 9, 6]$

For the right sub-array, choose 6 as the pivot (last element).
elements less than $6 \longrightarrow []$
elements greater than $6 \longrightarrow [7, 9, 8]$

Since, we don't have left sub-array.
We process right sub-array $[7, 9, 8]$

Now choose 8 as the pivot element (last element)

elements less than $8 \longrightarrow [7]$
elements greater than $8 \longrightarrow [9]$

After partitioning $[7, 9, 8]$ with 8 as the pivot element, we have,
left sub-array = $[7]$ and right sub-array = $[9]$
Since, now, only single elements are left. Now, we have reached base case, so we can stop here.
The left sub-array is now in sorted form.
So we get the sorted list $[0, 1]$

Similarly, for right sub-array $\longrightarrow$
sorted right sub-array = $[6, 7, 8, 9]$
So, the final sorted output array is $[0, 1, 2, 6, 7, 8, 9]$

3.7 Coding Problem

Question:
Implement quicksort algorithm on the following test cases:

1. test case 1 $= [5, 1, 1, 2, 0, 0, 9]$
2. test case 2 $= [5, 2, 7, 3, 1]$
3. test case 3 $= [9, 8, 7, 6, 1, 5, 4, 3]$

Solution:
Please refer the Quicksort Python file for the complete solution.

Chapter 4
Greatest Common Divisor

4.1 Introduction

What is GCD?

The greatest common factor that divides every number so that its remainder is zero is known as the Greatest Common Divisor (GCD), or Highest Common Factor (HCF), of two or more numbers.

Methods to find GCD $\longrightarrow$

Prime Factorization Method

Long Division Method

Euclidean Algorithm

In this chapter, we will specifically explore Euclidean Algorithm and Extended Euclidean Algorithm.

What is Euclidean Algorithm?

The Euclidean Algorithm, also known as the Euclidean Division Algorithm, simplifies finding the Greatest Common Divisor (GCD) of two integers. It breaks down the problem into smaller tasks, gradually converging toward the solution.

The Euclidean Algorithm, dating back to around 300 BC, relies on the principle that the Greatest Common Divisor (GCD) of two numbers remains unchanged when the larger number is replaced by its difference with the smaller number. For example:
$GCD(252, 105) = GCD(252 - 105, 105) = GCD(147, 105) = 21$

Euclid's Rule:

If x and y are positive integers with $x \geq y$, then $gcd(x, y) = gcd(x \mod y, y)$.

From the above paragraph, we have, $gcd(x, y) = gcd(x - y, y)$
An integer that divides x and y, must also divide $x - y$. So, $gcd(x, y) \leq gcd(x - y, y)$.

Similarly, any integer that divides both $x - y$ and y must also divide both x and y.
So, $gcd(x - y, y) \geq gcd(x, y)$.

4.2 Euclid's Algorithm

Let's see the pseudocode of Euclid's algorithm to find the gcd of two numbers using recursive approach.

```
function Euclid (a, b)
Input: Two integers a and b with $a \geq b \geq 0$
Output: gcd(a,b)

if $b = 0$:
return a

return Euclid(b, a mod b)
```

This method finds the greatest common divisor (GCD) of two integers, (a) and (b), using the Euclidean algorithm implemented in a recursive manner. The procedure carries out a function that accepts two integer parameters, (a) and (b), and outputs an integer. First, the method determines whether (b) equals 0. If so, it indicates that (a) is the GCD and that (a) is the outcome. But when (a) is divided by (b), if (b) is not 0, it means that there is a remainder. Here, the procedure uses the arguments (b) and $(a \mod b)$ to call itself repeatedly. Until (b) equals 0, this recursive call is made, which results in the base case and the GCD.

4.3 Extended Euclid's Algorithm

The extended Euclidean algorithm is an algorithm to compute integers x and y such that $ax + by = gcd(a, b)$.

We can think of the extended Euclidean algorithm as the modular exponentiation reciprocal.
These integers, x and y, can be found by reversing the steps in the Euclidean algorithm. The goal is to work backwards recursively, starting with the GCD. To do this, we can consider the numbers as variables until we get an equation that combines our starting numbers in a linear fashion.

Now, let's see the pseudocode of Extended Euclid's algorithm using recursive approach.

```
function Extended-Euclid (a, b)
Input: Two positive integers a and b with $a \geq b \geq 0$
Output: Integers x, y, d such that $d = gcd(a, b)$ and $ax + by = d$

if $b = 0$:
return $(1, 0, a)$
$(x', y', d)$ = Extended-Euclid $(b, a \mod b)$

return $(y', x' - [a/b]y', d)$
```

Using the results of the recursive call, $gcd(b \mod a, a)$, the extended Euclidean algorithm updates the results of gcd(a, b). Let x' and y' be the values of x and y that the recursive call calculated. We update x and y with the following expressions.

$ax + by = gcd(a, b)$
$gcd(a, b) = gcd(b \mod a, a)$
$gcd(b \mod a, a) = (b \mod a)x' + ay'$
$ax + by = (b \mod a)x' + ay'$
$ax + by = (b\breve{}[b/a] * a)x' + ay'$
$ax + by = a(y'\breve{}[b/a] * x') + bx'$

Comparing LHS and RHS,
x = y' – [b/a] * x'
y = x'

4.4 Time and Space Complexity Analysis

1. The Euclidean Algorithm has a time complexity of $\mathcal{O}(\log(min(a, b)))$.
   Here, a and b are two integers.
   Recursively, it's expressed in the form:
   $gcd(a, b) = gcd(b, a \mod b)$
   The time complexity can be described as the number of steps needed to reduce b to 0. The number of iterations is determined by the Fibonacci sequence, which grows logarithmically with respect to the input values.

2. The space complexity of the Euclidean Algorithm is also $\mathcal{O}(\log(min(a, b)))$. Here, auxiliary space is used for recursive call stack.

3. The time complexity for the Extended Euclidean Algorithm is also same i.e., $\mathcal{O}(\log(min(a, b)))$.
   Until the smaller number equals zero, the algorithm repeatedly divides the larger number by the smaller one.

4. The space complexity for the Extended Euclidean Algorithm is $\mathcal{O}(\log(min(a, b)))$. Here, auxiliary space is used for recursive call stack.

4.5 Exercise Problem

Question:
Use the Euclidean Algorithm to find the greatest common divisor of 44 and 17. Find integers x and y solving the equation $44x + 17y = 1$ with $x > 10$ using Extended Euclid's Algorithm.

Solution:
The Euclidean Algorithm yields the following steps:

Step 1: Divide 44 by 17
$44 = 2 * 17 + 10$

Step 2: We now divide the divisor (17) by the remainder (10) because the remainder equals 10:
$17 = 1 * 10 + 7$

Step 3: Then divide the new remainder (7) by the preceding remainder (10):
$10 = 1 * 7 + 3$

Continue the above step by dividing 7 by 3:
$7 = 2 * 3 + 1$

Therefore the greatest common divisor of 44 and 17 is 1 .

Now, to find x and y values, using Extended Euclid Algorithm,

we already have, GCD of 44 and 17 is 1.

By reversing the steps in the Euclidean algorithm, it is possible to find these integers x and y.

$1 = 7 - 2 * 3$
$1 = 7 - 2 * (10 - 7) = 3 * 7 - 2 * 10$
$1 = 3 * (17 - 10) - 2 * 10 = 3 * 17 - 5 * 10$

$1 = 3 * 17 - 5 * (44 - 2 * 17) = 13 * 17 - 5 * 44$

With this, we get, $x = 5$, $y = 13$.

But, we have to check the condition where $x > 10$

$-5 * 44 + 13 * 17 = 1$
$17 * 44 - 44 * 17 = 0$
Finally, $12 * 44 - 31 * 17 = 1$

Therefore, the integers $x = 12$ and $y = -31$ satisfy the equation with x greater than 10.

Reflection:
During my synthesis assignments, I developed a deeper understanding of essential concepts such as Divide-and-Conquer, Mergesort, Quicksort, and the Greatest Common Divisor (GCD). Additionally, I gained insights into solving problems related to quicksort, mergesort, GCD, binary multiplication using divide-and-conquer, and recurrence relations by recognizing patterns and applying the Master Theorem. Furthermore, I tackled coding problems to solidify my practical knowledge of these concepts.

Acknowledgements:
I would like to express my sincere gratitude to the following individuals and resources for their invaluable contributions to this assignment:
1) Prof. Bruce Maxwell: Thank you, Professor Bruce Maxwell for your guidance in this synthesis assignment. Your expertise in algorithms greatly influenced my work.
2) Classmates and TA's: I appreciate the discussions of my classmates, and TA's who clarified my doubts.
3) DPV Algorithms Textbook: This textbook was a useful resource for me to understand the basics of algorithms.
4) https://dev.to/downey/using-the-master-theorem-to-solve-recurrences-4pdb: This website provided clear explanations for solving recurrence relations using Master Theorem.
5) https://medium.com/karuna-sehgal/a-quick-explanation-of-quick-sort-7d8e2563629b: This website was useful to quickly understand the quicksort algorithm.
6) https://brilliant.org/wiki/extended-euclidean-algorithm/: This website was helpful to understand Euclid Algorithm.