Department of Computer Science and Design

TE (Computer Science and Design 2019 Course) Semester –I

Subject: Design and Analysis of Algorithms Lab

Manual Content

Program Outcomes:

PO1	**Engineering Knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and computer engineering to the solution of complex engineering problems.
PO2	**Problem analysis:** Identify, formulate, review research literature, and analyze complex computer engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/Development of solutions:** Design solutions for complex computer engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	**Conduct Investigations of complex problems:** Use research based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern Tool usage:** Create, select, and apply appropriate techniques, and modern engineering and IT tools including prediction and modeling to complex computer engineering activities with an understanding of the limitations.
PO6	**The engineer and society:** Apply reasoning informed by the contextual knowledge to assess society, health, safety, legal and cultural issues and the consequent responsibilities relevant to the computer engineering practice.
PO7	**Environment and Sustainability:** Understand the computer engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics:** Apply ethical principles and commit to computer engineering ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication:** Communicate effectively on complex computer engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11	**Project Management and Finance:** Demonstrate knowledge and understanding of computer engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in broadest context of technological change.

Program Specific Outcomes:

PSO1:	The ability to understand, analyze and develop computer programs in the areas related to Algorithms, System Software, Machine Learning, Artificial Intelligence, Web Applications, Big Data Analytics and Networking for efficient design of computer based systems of varying complexity.
PSO2:	The ability to apply standard practices and strategies in software / embedded project development using open source programming tools and environment to deliver a quality product for business success.
PSO3:	The ability to employ modern computer language, environment and platforms in creating innovative career paths to be an entrepreneur and path for higher studies.

Program Educational Outcomes (PEOs):

1. To work in a multidisciplinary environment by providing solutions to real time problems.
2. To develop computer programmers on design and programming techniques, technologies and tools related to computer engineering.
3. To inculcate, in students, professional and ethical attitude, effective communication skills, team work skill and ability to relate engineering issues in broader social context.

CO-PO AND CO-PSO MAPPING MATRIX

Class: TE Computer (2019 Pattern)	Sub: Design and Analysis of Algorithms Lab

Course Outcomes (CO): Course outcomes are the statements of what a student should know, understand and/or be able to demonstrate after completion of a course.

Course Outcomes	Description of COs
CO1	**CO1: Apply (L3: Apply)** algorithmic strategies for solving various problems
CO2	**CO2: Analyze (L4: Analyze) performance of an algorithm.**
CO3	**CO3: Implement (L3: Apply)** an algorithm that follows one of the following algorithm design strategies: divide and conquer, greedy, dynamic programming, backtracking, branch and bound.

CO-PO and CO-PSO Mapping Matrix:

Course Outcomes	PO 1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO 10	PO 11	PO 12	PSO1	PSO 2	PSO 3
C446.1	3	3	3	1	2	1	-	1	2	-	2	3	1	-	-
C446.2	3	3	3	2	2	1	-	1	2	-	2	3	1	-	-
C446.3	3	3	3	2	2	2	-	1	2	-	2	3	1	-	-

Guidelines

Guidelines for Instructor's Manual

The instructor’s manual is to be developed as a reference and hands-on resource. It should include prologue (about University/program/ institute/ department/foreword/ preface), curriculum of the course, conduction and assessment guidelines, topics under consideration, concept, objectives, outcomes, set of typical applications/assignments/ guidelines, and references.

Guidelines for Student's Laboratory Journal

The laboratory assignments are to be submitted by students in the form of a journal. Journal consists of Certificate, table of contents, and handwritten write-up of each assignment (Title, Date of Completion, Objectives, Problem Statement, Software and Hardware requirements, Assessment grade/marks and assessor's sign, Theory- Concept in brief, algorithm, flowchart, test cases, Test Data Set(if applicable), mathematical model (if applicable), conclusion/analysis. Program codes with sample output of all performed assignments are to be submitted as a softcopy. As a conscious effort and little contribution towards Green IT and environment awareness, attaching printed papers as part of write-ups and program listing to a journal must be avoided. Use of DVD containing student programs maintained by Laboratory In-charge is highly encouraged. For reference one or two journals may be maintained with program prints in the Laboratory.

Home

Guidelines for Laboratory /Term Work Assessment

Continuous assessment of laboratory work should be based on overall performance of Laboratory assignments by a student. Assessment of each Laboratory assignment will assign grade/marks based on parameters, such as timely completion, performance, innovation, efficient codes, punctuality, documentation and neatness.

Guidelines for Practical Examination

Problem statements must be decided jointly by the internal examiner and external examiner. During practical assessment, maximum weightage should be given to satisfactory implementation of the problem statement. Relevant questions may be asked at the time of evaluation to test the student’s understanding of the fundamentals, effective and efficient implementation. This will encourage, transparent evaluation and fair approach, and hence will not create any uncertainty or doubt in the minds of the students. So, adhering to these principles will consummate our team efforts to the promising start of student's academics.

Guidelines for Laboratory Conduction

The instructor is expected to frame the assignments by understanding the prerequisites, technological aspects, utility and recent trends related to the topic. The assignment framing policy needs to address the average students and inclusive of an element to attract and promote the intelligent students. Use of open source software is encouraged. Based on the concepts learned. Instructors may also set one assignment or mini-project that is suitable to each branch beyond the scope of the syllabus.

Operating System recommended: - 64-bit Open source Linux or its derivative Programming tools recommended: - C++, Java, Python, Solidity, etc.

List of Laboratory Experiments/Assignments.	
Design and Analysis of Algorithms	
1	Write a program to calculate Fibonacci numbers and find its step count.
2	Implement job sequencing with deadlines using a greedy method.
3	Write a program to solve a fractional Knapsack problem using a greedy method.
4	Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.
5	Write a program to solve the travelling salesman problem and to print the path and the cost using Branch and Bound.
6	Design 8-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final 8-queen's matrix.
7	Write a program for analysis of quick sort by using deterministic and randomized variant
8	Mini Project: Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also analyze the performance of each algorithm for the best case and the worst case.

Experiment No. 1

Write a program to calculate Fibonacci numbers and find its step count.

Aim: Write a program non-recursive and recursive program to calculate Fibonacci numbers.

Theory:

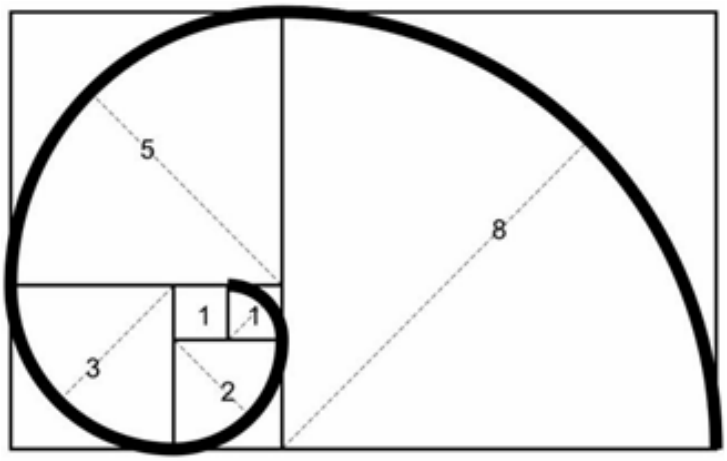
- **Introduction to Fibonacci numbers**
  - The Fibonacci series, named after Italian mathematician Leonardo Pisano Bogollo, later known as Fibonacci, is a series (sum) formed by Fibonacci numbers denoted as Fn. The numbers in Fibonacci sequence are given as: 0, 1, 1, 2, 3, 5, 8, 13, 21, 38, …
  - In a Fibonacci series, every term is the sum of the preceding two terms, starting from 0 and 1 as first and second terms. In some old references, the term '0' might be omitted.
- **What is the Fibonacci Series?**
  - The Fibonacci series is the sequence of numbers (also called Fibonacci numbers), where every number is the sum of the preceding two numbers, such that the first two terms are '0' and '1'.
  - In some older versions of the series, the term '0' might be omitted. A Fibonacci series can thus be given as, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . . It can thus be observed that every term can be calculated by adding the two terms before it.
  - Given the first term, F0 and second term, F1 as '0' and '1', the third term here can be given as, F2 = 0 + 1 = 1

    Similarly,

    F3 = 1 + 1 = 2

    F4 = 2 + 1 = 3

Given a number n, print n-th Fibonacci Number



Algorithm:

Algorithm:

- **Step 1:** Start
- **Step 2:** Declare variables i, a, b , show
- **Step 3:** Initialize the variables, a=0, b=1, and show =0
- **Step 4:** Enter the number of terms of Fibonacci series to be printed
- **Step 5:** Print First two terms of series
- **Step 6:** Use loop for the following steps

  show=a+b

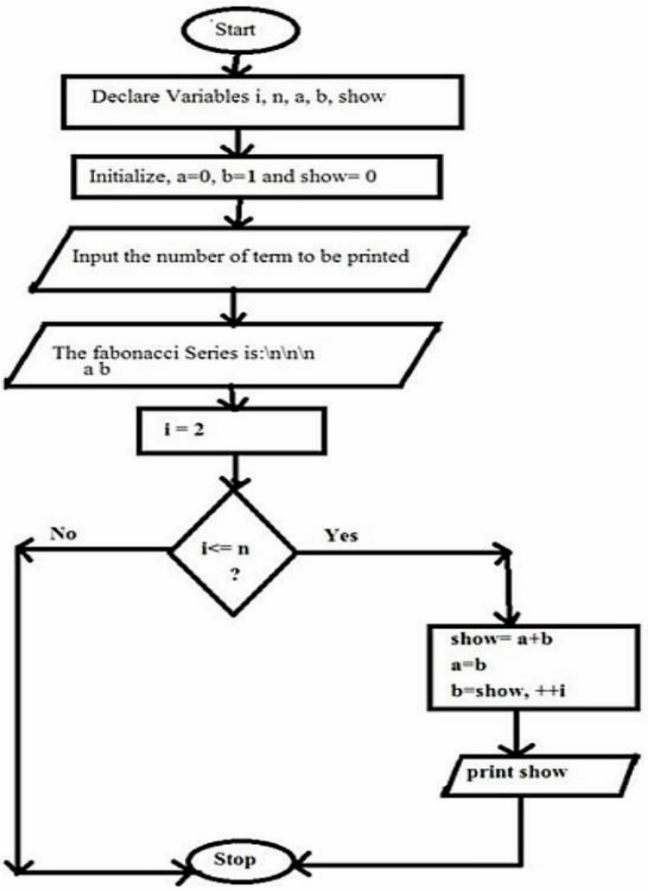
  a=b

  b=show

  Increase value of i each time by 1

  print the value of show
- **Step 7:** End

Flow Chart:



Conclusion: Here we learned the Fibonacci series, and performed a program on it to find out the complexity of recursive and non-recursive functions.

Experiment No. 2.

Implement job sequencing with deadlines using a greedy method.

Aim: Write a program to implement Huffman Encoding using a greedy strategy.

Theory:

- **Huffman Encoding**

  Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.
  Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

  Huffman Coding is a famous Greedy Algorithm.

  It is used for the lossless compression of data.

  It uses variable length encoding.

  It assigns variable length code to all the characters.

  The code length of a character depends on how frequently it occurs in the given text.

  The character which occurs most frequently gets the smallest code.

  The character which occurs least frequently gets the largest code.

  It is also known as Huffman Encoding.

- **Job Sequencing With Deadlines:**

  The sequencing of jobs on a single processor with deadline constraints is called Job Sequencing with Deadlines.

  Here:

  - You are given a set of jobs.
  - Each job has a defined deadline and some profit associated with it.
  - The profit of a job is given only when that job is completed within its deadline.
  - Only one processor is available for processing all the jobs.
  - Processor takes one unit of time to complete a job.

- **The problem states:**

"How can the total profit be maximized if only one job can be completed at a time?"

- **Approach to Solution:**

A feasible solution would be a subset of jobs where each job of the subset gets completed within its deadline.

Value of the feasible solution would be the sum of profit of all the jobs contained in the subset.

An optimal solution of the problem would be a feasible solution which gives the maximum profit.

Greedy Algorithm:

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.

The greedy algorithm described below always gives an optimal solution to the job sequencing problem:

Sort all the given jobs in decreasing order of their profit. Check the value of the maximum deadline.

Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.

Pick up the jobs one by one.

Put the job on the Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

Conclusion:

Hence, here we studied what is job scheduling and what is deadlock. And we learned that how to handle deadlock using greedy method

Experiment No. 3.

Write a program to solve a fractional Knapsack problem using a greedy method.

Aim: Write a program to solve a fractional Knapsack problem using a greedy method.

Theory:

Greedy Method:

- A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.
- The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top- down approach.
- This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

Advantages of Greedy Approach

- The algorithm is easier to describe.
- This algorithm can perform better than other algorithms (but, not in all cases).

Drawback of Greedy Approach

- As mentioned earlier, the greedy algorithm doesn't always produce the optimal solution. This is the major disadvantage of the algorithm
- For example, suppose we want to find the longest path in the graph below from root to leaf.

Greedy Algorithm

1. To begin with, the solution set (containing answers) is empty.
2. At each step, an item is added to the solution set until a solution is reached.
3. If the solution set is feasible, the current item is kept.

Knapsack Problem

- You are given the following:
  - A knapsack (kind of shoulder bag) with limited weight capacity.
  - Few items each having some weight and value.

The problem states:

- Which items should be placed into the knapsack such that
  - The value or profit obtained by putting the items into the knapsack is maximum.
  - And the weight limit of the knapsack does not exceed.

Knapsack Problem Variants:

- Knapsack problem has the following two variants:
  - 1. Fractional Knapsack Problem
  - 2. 0/1 Knapsack Problem

Fractional Knapsack Problem:

- In Fractional Knapsack Problem,
  - As the name suggests, items are divisible here.
  - We can even put the fraction of any item into the knapsack if taking the complete item is not possible.
  - It is solved using the Greedy Method.

Fractional Knapsack Problem Using Greedy Method:

- Fractional knapsack problem is solved using greedy method in the following steps-
  - **Step-01:** For each item, compute its value / weight ratio.
  - **Step-02:** Arrange all the items in decreasing order of their value / weight ratio.
  - **Step-03:**

    Start putting the items into the knapsack beginning from the item with the highest ratio. Put as many items as you can into the knapsack.

Time Complexity:

The main time taking step is the sorting of all items in decreasing order of their value / weight ratio.
If the items are already arranged in the required order, then the loop takes O(n) time.

The average time complexity of Quick Sort is O(n log n).

Conclusion:

Hence, here we studied the Greedy Method, Greedy Algorithm, Knapsack, Knapsack Problem.

Experiment No. 4.

Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

Aim: Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

Theory:

- **Dynamic Programming:**
  - Dynamic Programming is also used in optimization problems. Like divide-and-conquer methods, Dynamic Programming solves problems by combining the solutions of subproblems.
  - Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.
  - Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are overlapping sub-problems and optimal substructure.
  - Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exist.
  - For example, Binary Search does not have overlapping sub-problem. Whereas recursive programs of Fibonacci numbers have many overlapping subproblems.
- **Steps of Dynamic Programming Approach**

  Dynamic Programming algorithm is designed using the following four steps:
  - Characterize the structure of an optimal solution.
  - Recursively define the value of an optimal solution.
  - Compute the value of an optimal solution, typically in a bottom-up fashion.
  - Construct an optimal solution from the computed information.
- **Applications of Dynamic Programming Approach**
  - Matrix Chain Multiplication
  - Longest Common Subsequence
  - Travelling Salesman Problem

- **Knapsack Problem**

  You are given the following:

  - A knapsack (kind of shoulder bag) with limited weight capacity.
  - Few items each having some weight and value.

- **The problem states:**
- Which items should be placed into the knapsack such that:
  - The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.

- **Knapsack Problem Variants**

  Knapsack problem has the following two variants:

  - Fractional Knapsack Problem
  - 0/1 Knapsack Problem

Conclusion: Hence, we solved a 0-1 Knapsack problem using dynamic programming or branch and bound strategy in the program.

Experiment No. 5.

Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen's matrix

Aim: Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen's matrix.

Theory:

Backtracking:

Backtracking is finding the solution of a problem whereby the solution depends on the previous steps taken.

For example, in a maze problem, the solution depends on all the steps you take one-by-one. If any of those steps is wrong, then it will not lead us to the solution. In a maze problem, we first choose a path and continue moving along it. But once we understand that the particular path is incorrect, then we just come back and change it. This is what backtracking basically is.

In backtracking, we first take a step and then we see if this step taken is correct or not i.e., whether it will give a correct answer or not. And if it doesn't, then we just come back and change our first step. In general, this is accomplished by recursion. Thus, in backtracking, we first start with a partial sub-solution of the problem (which may or may not lead us to the solution) and then check if we can proceed further with this sub-solution or not. If not, then we just come back and change it.

Thus, the general steps of backtracking are:

- start with a sub-solution
- check if this sub-solution will lead to the solution or not
- If not, then come back and change the sub-solution and continue again

Applications of Backtracking:

- N Queens Problem
- Sum of subsets problem
- Graph coloring
- Hamiltonian cycles.

N queens on NxN chessboard

One of the most common examples of the backtracking is to arrange N queens on an NxN chessboard such that no queen can strike down any other queen. A queen can attack horizontally,

vertically, or diagonally. The solution to this problem is also attempted in a similar way. We first place the first queen anywhere arbitrarily and then place the next queen in any of the safe places. We continue this process until the number of unplaced queens becomes zero (a solution is found) or no safe place is left. If no safe place is left, then we change the position of the previously placed queen.

- **N-Queens Problem:**

  A classic combinational problem is to place n queens on a n*n chess board so that no two attack, i.,e no two queens are on the same row, column or diagonal.

What is the N Queen Problem?

N Queen problem is the classical Example of backtracking. N-Queen problem is defined as,

"given N x N chess board, arrange N queens in such a way that no two queens attack each other by being in the same row, column or diagonal".

For N = 1, this is a trivial case.

For N = 2 and N = 3, a solution is not possible.

So we start with N = 4 and we will generalize it for N queens.

- If we take n=4 then the problem is called the 4 queens problem.
- If we take n=8 then the problem is called the 8 queens problem.

Algorithm:

Step 1: Start in the leftmost column

Step 2: If all queens are place return true

Step 3: Try all rows in the current column.

Do the following for every tried row.

a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
b) If placing the queen in [row, column] leads to a solution then return true.
c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.

Conclusion:

Hence, we learn backtracking, and create a N Queen Problem solver. For example we tested 8 queens chess boards.

Experiment No. 6.

Mini Project

Aim: Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also analyse the performance of each algorithm for the best case and the worst case

Theory:

Merge Sort Algorithm

The Merge Sort algorithm is a sorting algorithm that is based on the Divide and Conquer paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

Merge Sort Working Process:

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

Merge Sort using Multi-threading

Merge Sort is a popular sorting technique which divides an array or list into two halves and then starts merging them when sufficient depth is reached. Time complexity of merge sort is O(n log n).

Threads are lightweight processes and threads shared with other threads, their code section, data section and OS resources like open files and signals. But, like a process, a thread has its own program counter (PC), a register set, and a stack space.

Multi-threading is a way to improve parallelism by running the threads simultaneously in different cores of your processor. In this program, we'll use 4 threads but you may change it according to the number of cores your processor has.

Algorithm:

Step 1: **Start**

Step 2: **declare array and left, right, mid variable**

Step 3: **perform merge function.**

if left > right return

mid= (left+right)/2

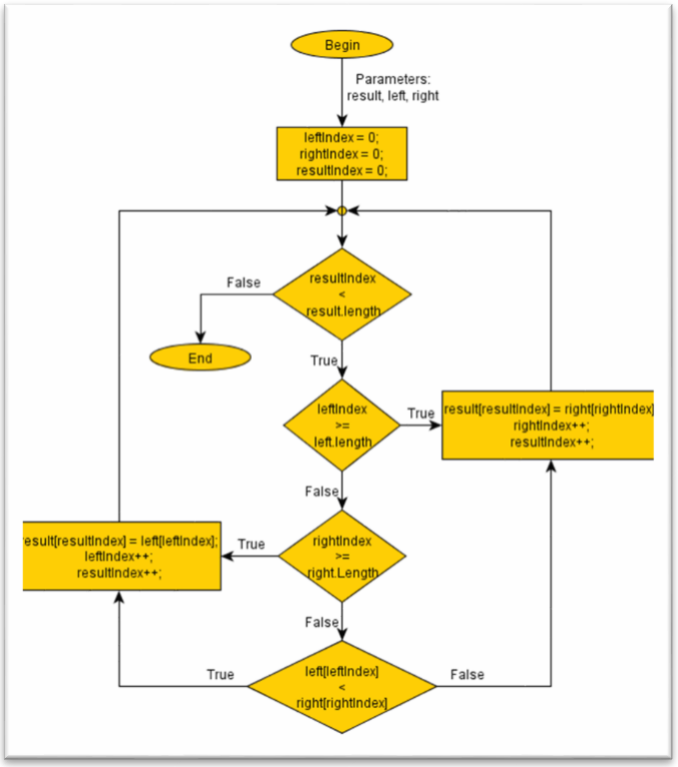
mergesort(array, left, mid)

mergesort(array, mid+1, right)

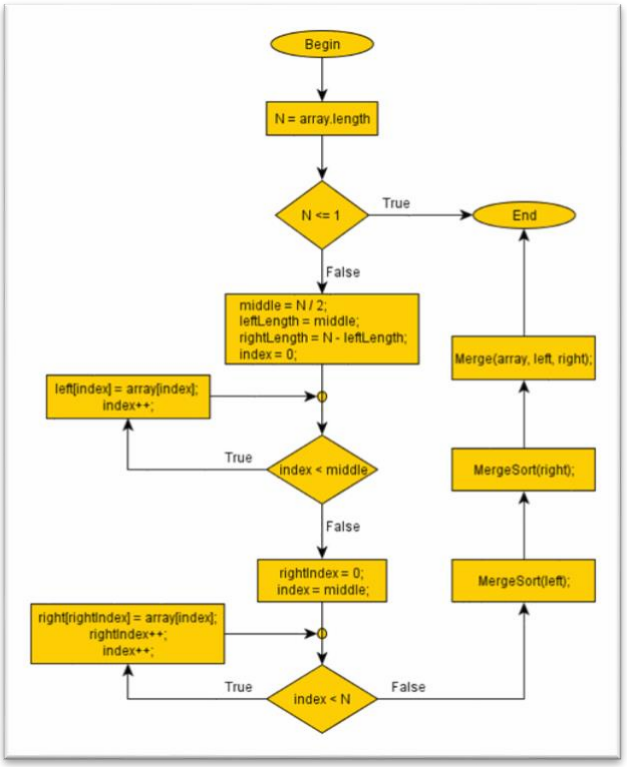
merge(array, left, mid, right)

Step 4: **Stop**

Flow Chart:



*** Merge routine ***



*** Merge sort ***

Program:

@contextmanager

def process_pool(size):

"""Create a process pool and block until all processes have completed."""

def para_merge_sort(array, ps_count): """Perform parallel

merge sort."""" timer = Timer("sort", "merge", "total")

timer.start_for("total") timer.start_for("sort")

step = int(length / ps_count) # Divide the list in chunks

Instantiate a multiprocessing.Manager obj to store the output manager = Manager()

res = manager.list()

```
        with process_pool(size=ps_count) as pool: for i in
            range(ps_count):
                # We create a new Process object and assign the #
                merge_sort_multiple function to it,
                # using as input a sublist if i <
                ps_count - 1:
                    chunk = array[i * step : (i + 1) * step] else:
                    # Get the remaining elements in the list chunk = array[i *
                    step :]
                pool.apply_async(merge_sort_multiple, (res, chunk))
        timer.stop_for("sort") print("Performing final
        merge.") timer.start_for("merge")
        # For a core count greater than 2, we can use multiprocessing # again to merge sub-lists in
        parallel.
        while len(res) > 1:
            with process_pool(size=ps_count) as pool:
                pool.apply_async(multMerge, (res, res.pop(0), res.pop(0)))
        timer.stop_for("merge")
        timer.stop_for("total") final_sorted_list =
        res[0] return timer, final_sorted_lis
def get_command_line_parameters():
    """Get the process count from command line parameters.""" if len(sys.argv) > 1:
        total_processes = int(sys.argv[1]) if total_processes
        > 1:
            # Restrict process count to even numbers if total_processes
            % 2 != 0:
                print("Process count should be an even number.") sys.exit(1)
        print(f"Using {total_processes} cores") else:
```

```
                total_processes = 1 return {"pc":
        total_processes}

if __name__ == " main ":
            -     -     -
        parameters = get_command_line_parameters()

        pc = parameters["pc"]

        main_timer = Timer("single_core", "list_generation") main_timer.start_for("list_generation")

        length = random.randint(3 * 10**6, 4 * 10**6)

        randArr=[random.randint(0,i*100) for i in range(length)] main_timer.stop_for("list_generation")

        print(f"List length: {length}")

        print(f"Random generated in {main_timer['list_generation']:.6f}")

        main_timer.start_for("single_core") single =
        merge_sort(randArr)
        main_timer.stop_for("single_core")

        randArr_sorted = randArr[:] # Create a copy first due to mutation randArr_sorted.sort()

        print("Verification of sorting algo:", randArr_sorted == single) print(f"Single Core elapsed time:
        {main_timer['single_core']:.6f} sec") print("Starting parallel sort.")

        para_timer, para_sorted_list = para_merge_sort( randArr, pc

        )print(f"Final merge duration: {para_timer['merge']:.6f} sec") print("Sorted arrays equal:",
        para_sorted_list == randArr_sorted) print(f"{pc}-Core elapsed time: {para_timer['total']:.6f} sec")
```

Output:

Using 16 cores

List length: 3252321

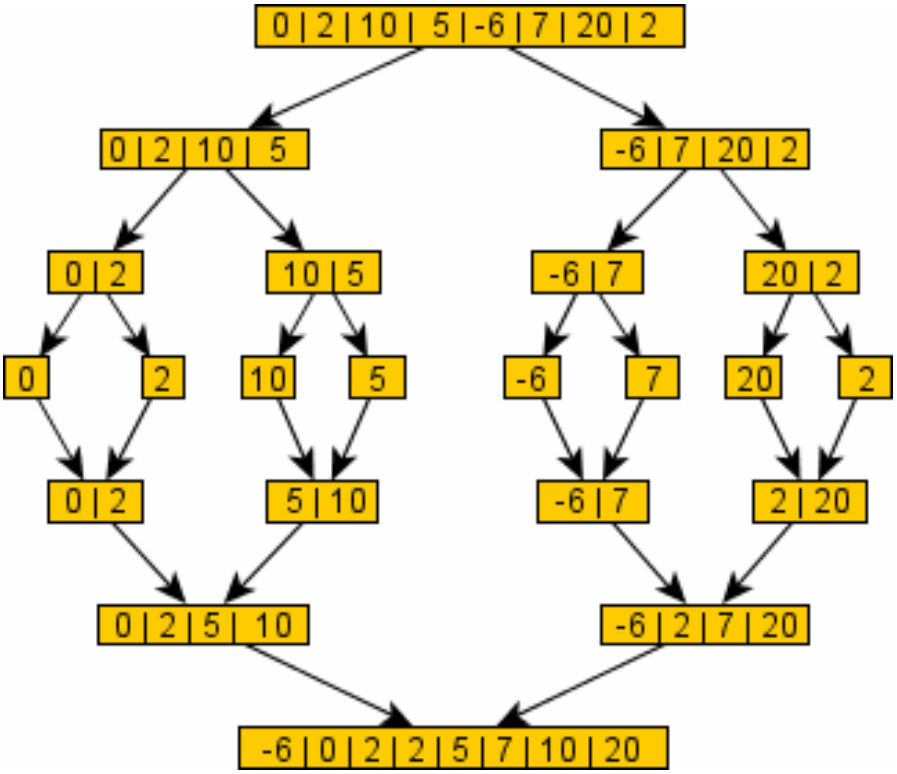
Random list generated in 1.787607 Verification of sorting
algorithm: True Single Core elapsed time: 14.204555 sec
Starting parallel sort.

Performing final merge.

Final merge duration: 4.764258 sec Sorted arrays
equal: True

16-Core elapsed time: 6.984671 sec

Idea Chart:



Conclusion:

Hence, we learned in this project are Sort, Merge Sort, Multi thread, Multi treated merge sort. We also calculated the time required.