

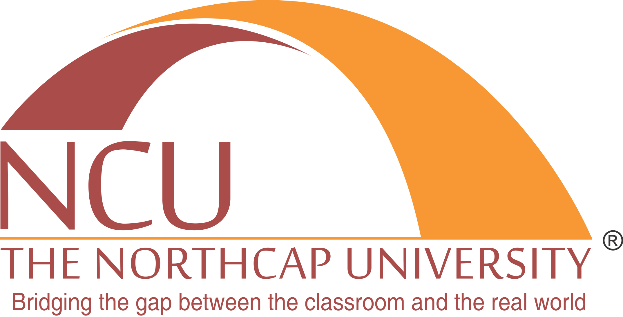
|  |  |
| --- | --- |
| **ANALYSIS AND DESIGN OF ALGORITHMS** | |
| Lab Manual | |
| **Department of Computer Science and Engineering**  **The NorthCap University, Gurugram** | |
|  |

**ANALYSIS AND DESIGN OF ALGORITHMS**

**Lab Manual**

**CSL 230**

**Mr. Sandeep Singh**



Department of Computer Science and Engineering

NorthCap University, Gurugram- 122001, India

Session 2019-20

*Published by:*

**School of Engineering and Technology**

**Department of Computer Science & Engineering**

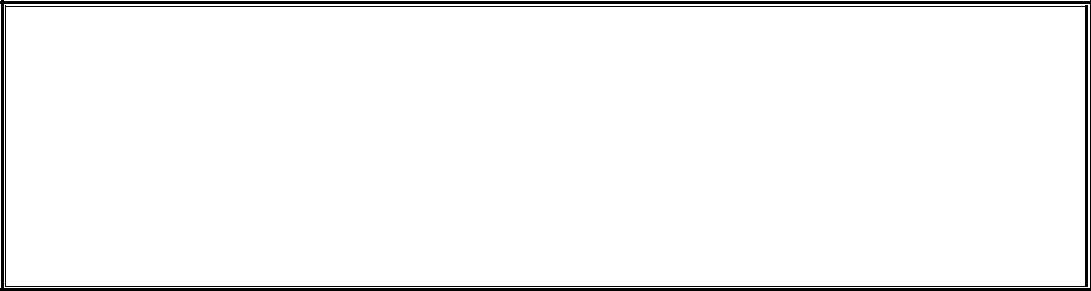
**The NorthCap University Gurugram**

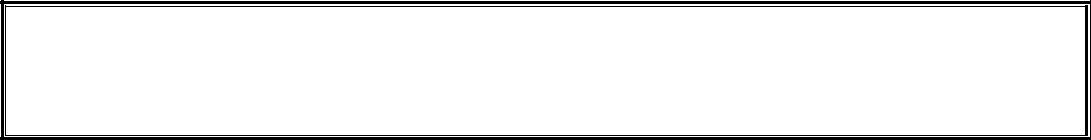
**• Laboratory Manual is for Internal Circulation only**

**© Copyright Reserved**

*No part of this Practical Record Book may be*

*reproduced, used, stored without prior permission of The NorthCap University*



Copying or facilitating copying of lab work comes under cheating and is considered as use of unfair means. Students indulging in copying or facilitating copying shall be awarded zero marks for that particular experiment. Frequent cases of copying may lead to disciplinary action. Attendance in lab classes is mandatory.

Labs are open up to 7 PM upon request. Students are encouraged to make full use of labs beyond normal lab hours.

**PREFACE**

Analysis and Design of Algorithms Lab Manual is designed to meet the course and program requirements of NCU curriculum for B.Tech 2nd year students of CSE branch. The concept of the lab work is to give brief practical experience for basic lab skills to students. It provides the space and scope for self-study so that students can come up with new and creative ideas.

The Lab manual is written on the basis of “teach yourself pattern” and expected that students who come with proper preparation should be able to perform the experiments without any difficulty. Brief introduction to each experiment with information about self-study material is provided. The laboratory exercises will familiarize the students with basic concepts of algorithm development and programming skills. The exercises include designing and analysis of basic algorithms like sorting and searching and will gradually develop programs for advanced techniques such as dynamic programming and greedy algorithms. The lab exercises will also enable the students to gain insights to advanced graph algorithms such as minimum spanning trees and shortest paths, NP-completeness theory. Students are expected to come thoroughly prepared for the lab. General disciplines, safety guidelines and report writing are also discussed.

The lab manual is a part of curriculum for The NorthCap University, Gurugram. Teacher’s copy of the experimental results and answer for the questions are available as sample guidelines.

We hope that lab manual would be useful to students of CSE branch and author requests the readers to kindly forward their suggestions / constructive criticism for further improvement of the work book.

Author expresses deep gratitude to Members, Governing Body-NCU for encouragement and motivation.

**Authors**

**The NorthCap University**

**Gurugram, India**

|  |  |  |  |
| --- | --- | --- | --- |
| **CONTENTS** | | | |
| S.N. | **Details** | Page No. | |
|  | **Syllabus** | 6 | |
| 1 | **Introduction** | 9 | |
| 2 | **Lab Requirement** | 10 |
| 3 | **General Instructions** | 11 |
| 4 | **List of Experiments** | 13 |
| 5 | **List of Flip Assignment** | 16 |
| 6 | **List of Projects** | 17 |
| 7 | **Rubrics** | 18 |
| 8 | **Annexure 1 (Format of Lab Report)** | 19 |
| 9 | **Annexure 2 (Format of Project Report)** | 20 |

**SYLLABUS**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1. **Department:** | | | | **Department of Computer Science and Engineering** | | | | | | | |
| 1. **Course Name:** Analysis and Design of Algorithms | | | | | | 1. **Course Code** | 1. **L-T-P** | | | | 1. **Credits** |
| CSL230 | 3-0-2 | | | | 4 |
| 1. **Type of Course (Check one):** | | | | Programme Core  **✓**  Programme Elective Open Elective | | | | | | | |
| 1. **Pre-requisite(s), if any:** Algebra, Programming Language | | | | | | | | | | | |
| 1. **Frequency of offering (check one):**   **✓✓✓✓**  Odd  Even **✓** Either semester Every semester | | | | | | | | | | | |
| 1. **Brief Syllabus:**   **This course is an introduction to analysis of algorithms. The course will start with designing and analysis of basic algorithms like sorting and searching and will gradually cover advanced techniques such as dynamic programming and greedy algorithms. Throughout the course, you will gain insights to advanced graph algorithms such as minimum spanning trees and shortest paths, NP-completeness theory. At the end of this course, students will be able to design algorithms for various computing problems and analyze the time and space complexity of algorithms. Students will be able to Critically analyze the different algorithm design techniques for a given problem as well as modify existing algorithms to improve efficiency** | | | | | | | | | | | |
| **Total lecture, Tutorial and Practical Hours for this course**  **(Take 15 teaching weeks per semester): 75** hours  The class size is maximum 30 learners. | | | | | | | | | | | |
| **Lectures: 45** hours | | | | | **Practice** | | | | | | |
| **Tutorials : 1**0 hours | | | | | **Lab Work: 2**0 hours | |
| 1. **Course Outcomes (COs)**   Possible usefulness of this course after its completion i.e., how this course will be practically useful to him once it is completed. | | | | | | | | | | | |
| **CO 1** | Design and analysis of algorithms for a given problem. | | | | | | | | | | |
| **CO 2** | Analyse various complexity measures and the performance of algorithms. | | | | | | | | | | |
| **CO 3** | Apply and analyse the complexity of certain divide and conquer, greedy, and dynamic programming algorithms. | | | | | | | | | | |
| **CO 4** | Explain and apply backtracking algorithms. | | | | | | | | | | |
| **CO 5** | Ability to design and analyse branch and bound techniques to deal with some hard problems. | | | | | | | | | | |
| **CO 6** | Understand the classes P, NP, and NP-Complete and be able to prove that a certain problem is NP-Complete. | | | | | | | | | | |
| 1. **UNIT WISE DETAILS No. of Units: 6** | | | | | | | | | | | |
| **Unit Number: 1** | | **Title: Introduction to algorithms** | | | | | | **No. of hours: 7** | | | |
| **Content Summary:**  Role of algorithms in computing, Algorithms as technology, analyzing and designing algorithms, Growth of Functions, Asymptotic notations, Recurrences, Substitution method, Recursion tree method, Master method. | | | | | | | | | | | |
| **Unit Number: 2** | | **Title: Divide and Conquer** | | | | | | **No. of hours: 7** | | | |
| **Content Summary:**  General method, binary search, merge sort, quick sort, selection sort, heap sort, insertion sort, Stassen’s matrix multiplication algorithms and analysis of algorithms for these problems. | | | | | | | | | | | |
| **Unit Number: 3** | | **Title: Greedy and Dynamic Programming** | | | | | | **No. of hours: 15** | | | |
| **Content Summary:**  General method, knapsack problem, job sequencing with deadlines, minimum spanning trees (Kruskal's Algorithm, Prim's Algorithm), Shortest path algorithm (Dijkstra's Algorithm) and analysis of these problems. BFS, DFS, Activity selection problem. Dynamic Programming: General method, Principle of optimality, 0/1- knapsack, the traveling salesperson problem, Optimal binary search tree. | | | | | | | | | | | |
| **Unit Number: 4** | | **Title: Backtracking** | | | | | | **No. of hours: 8** | | | |
| **Content Summary:**  General method, 8-queen’s problem, subset sum problem, Graph Coloring, Hamiltonian cycles, analysis of these problems. | | | | | | | | | | | |
| **Unit Number: 5** | | **Title: Branch and Bound** | | | | | | **No. of hours: 5** | | | |
| **Content Summary:**  Introduction to Branch and Bound, LC search and FIFO search, 0/1- knapsack and traveling salesperson problem, efficiency considerations. | | | | | | | | | | | |
| **Unit Number: 6** | | | **Title: NP and NP complete** | | | | | | **No. of hours: 3** | | |
| **Content Summary:** Basic concepts, Cook’s theorem, NP hard graph and NP scheduling problems some simplified NP hard problems. | | | | | | | | | | | |
| 1. **Brief Description of Self-learning components by students (through books/resource material etc.):**   Supplementary MOOC Courses  <https://onlinecourses.nptel.ac.in/noc21_cs22/preview>  [Design and Analysis of algorithms - Course (swayam2.ac.in)](https://onlinecourses.swayam2.ac.in/cec20_cs03/preview)  GATE/NET/other PSU Exams  [Algorithms | Subject Wise Questions – AcademyEra](http://academyera.com/algorithms)  [Gate CSE Question Bank – AcademyEra](https://academyera.com/gate-cse-question-bank)  <https://www.geeksforgeeks.org/gate-corner-2-gq/>  [Algorithm (gradeup.co)](https://gradeup.co/computer-science-engineering-l-algorithm-i-ijpjhg2d) | | | | | | | | | | | |
| 1. **Books Recommended :**   **Textbooks:**   1. Ellis Horowitz, Sartaj Sahani, Sanguthevar Rajashekaran, “Fundamentals of Computer Algorithms”, Orient Black Swan, 2nd Edition, 2008. 2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, “Introduction to Algorithms”, MIT Press, 3rd Edition, 2009.   **Reference Books:**   1. J. Kleinberg and E. Tardos, “Algorithm Design”, Pearson,1st Edition, 2013.   **Reference Websites: (nptel, swayam, coursera, edx, udemy, lms, official documentation weblink)**  <https://swayam.gov.in/nd1_noc20_cs10/preview>  [The Design and Analysis of Algorithm Masterclass [ 2019 ] | Udemy](https://www.udemy.com/course/design-and-analysis-of-algorithm-/)  [Design and Analysis of Algorithms | Udemy](https://www.udemy.com/course/design-and-analysis-of-algorithms/)  **eBooks:**   * [Horowitz and Sahani, Fundamentals of Computer Algorithms, 2ND Edition - PDF Drive](https://www.pdfdrive.com/horowitz-and-sahani-fundamentals-of-computer-algorithms-2nd-edition-e18723362.html) * [Introduction to Algorithms, Third Edition - PDF Drive](https://www.pdfdrive.com/introduction-to-algorithms-third-edition-e42987274.html) * [Algorithms illuminated Part 2 Graph Algorithms and Data Structures by Tim Roughgarden - PDF Drive](https://www.pdfdrive.com/algorithms-illuminated-part-2-graph-algorithms-and-data-structures-e158470397.html) * [Python Algorithms: Mastering Basic Algorithms by Magnus Lie Hetland - PDF Drive](https://www.pdfdrive.com/python-algorithms-mastering-basic-algorithms-e31889331.html)   **Interview/Placement related Commonly asked Questions:**   * [Top 8 Algorithm Interview Questions And Answer {Updated For 2020} (educba.com)](https://www.educba.com/algorithm-interview-questions/) * [Top 25 Algorithm Interview Questions - javatpoint](https://www.javatpoint.com/algorithm-interview-questions) * [19 Essential Algorithm Interview Questions and Answers (toptal.com)](https://www.toptal.com/algorithms/interview-questions) * [Commonly Asked Algorithm Interview Questions | Set 1 - GeeksforGeeks](https://www.geeksforgeeks.org/commonly-asked-algorithm-interview-questions-set-1/) | | | | | | | | | | | |

1. **INTRODUCTION**

That ‘learning is a continuous process’ cannot be over emphasized. The theoretical knowledge gained during lecture sessions need to be strengthened through practical experimentation. Thus, practical makes an integral part of a learning process.­­­­­­­­­­­­­­­­­­­­­­­ The algorithms learnt during the theory classes will be implemented in the lab sessions which will help the student to build strong and deeper understanding of the concepts.

**COURSE OBJECTIVES:**

1. Design and analysis of algorithms for a given problem.
2. Analyse various complexity measures and the performance of algorithms.
3. Apply and analyse the complexity of certain divide and conquer, greedy, and dynamic programming algorithms.
4. Explain and apply backtracking algorithms.
5. Ability to design and analyse branch and bound techniques to deal with some hard problems.
6. Understand the classes P, NP, and NP-Complete and be able to prove that a certain problem is NP-Complete.
7. **LAB REQUIREMENTS**

|  |  |
| --- | --- |
| **Requirements** | **Details** |
| **Software Requirements** | C/C++/Java |
| **Operating System** | Any Operating System |
| **Hardware Requirements** |  |
| **Required Bandwidth** | NA |

1. **GENERAL INSTRUCTIONS** 
   1. **General discipline in the lab**
   * Students must turn up in time and contact concerned faculty for the experiment they are supposed to perform.
   * Students will not be allowed to enter late in the lab.
   * Students will not leave the class till the period is over.
   * Students should come prepared for their experiment.
   * Experimental results should be entered in the lab report format and certified/signed by concerned faculty/ lab Instructor.
   * Students must get the connection of the hardware setup verified before switching on the power supply.
   * Students should maintain silence while performing the experiments. If any necessity arises for discussion amongst them, they should discuss with a very low pitch without disturbing the adjacent groups.
   * Violating the above code of conduct may attract disciplinary action.
   * Damaging lab equipment or removing any component from the lab may invite penalties and strict disciplinary action.
   1. **Attendance**

* Attendance in the lab class is compulsory.
* Students should not attend a different lab group/section other than the one assigned at the beginning of the session.
* On account of illness or some family problems, if a student misses his/her lab classes, he/she may be assigned a different group to make up the losses in consultation with the concerned faculty / lab instructor. Or he/she may work in the lab during spare/extra hours to complete the experiment. No attendance will be granted for such case**.**
  1. **Preparation and Performance**
* Students should come to the lab thoroughly prepared on the experiments they are assigned to perform on that day. Brief introduction to each experiment with information about self-study reference is provided on LMS.
* Students must bring the lab report during each practical class with written records of the last experiments performed complete in all respect.
* Each student is required to write a complete report of the experiment he has performed and bring to lab class for evaluation in the next working lab. Sufficient space in work book is provided for independent writing of theory, observation, calculation and conclusion.
* Students should follow the Zero tolerance policy for copying / plagiarism. Zero marks will be awarded if found copied. If caught further, it will lead to disciplinary action.
* Refer **Annexure 1** for Lab Report Format.

1. **LIST OF EXPERIMENTS**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sr. No.** | | **Title of the Experiment** | **Software/Hardware based** | **Unit covered** | **Time Required** |
|  | | Implement and Calculate the time and space complexity of following programs:   1. Factorial of a number 2. Fibonacci series | C/C++/Java | 1 | 1 hour |
|  | | Implement Insertion/Selection sort algorithm and compute its time and space complexities. | C/C++/Java | 2 | 2 hours |
|  | | Implement Merge sort/Quick sort algorithm. Compute its time and space complexities | C/C++/Java | 2 | 2 hours |
|  | | Implement Strassen’s matrix algorithm. Compute its time complexity. | C/C++/Java | 2 | 1 hour |
|  | | Implement fractional knapsack algorithm. | C/C++/Java | 3 | 2 hours |
|  | | Implement Prim’s and Kruskal’s algorithms. Compute and compare their space and time complexities. | C/C++/Java | 3 | 2 hours |
|  | | Implement 0/1 Knapsack algorithm. | C/C++/Java | 3 | 2 hours |
|  | | Implement travelling salesman algorithm. | C/C++/Java | 4 | 2 hours |
| **Value Added Experiments** | | | | | |
|  | Implement n Queen’s Problem | | C/C++/Java | 4 | 2 hours |
|  | Implement Hamiltonian and Graph coloring algorithm. | | C/C++/Java | 4 | 2 hours |
|  | Implement 0/1 Knapsack problem using Branch and Bound | | C/C++/Java | 5 | 2 hours |

1. **FLIP EXPERIMENTS**
2. **LIST OF PROJECTS: NA**
3. **RUBRICS**

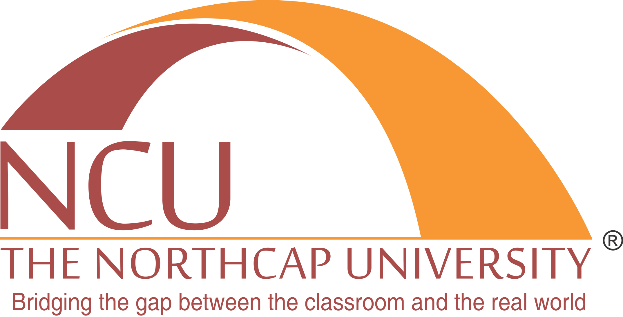
|  |  |
| --- | --- |
| **Marks Distribution** | |
| **Total** | **70** |
| **Online course(NPTEL)** | **10** |
| **Regular evaluation/ file/ mid term** | **30** |
| **Create GUI for Demonstration of any graph based algorithm/ Hackerank problem** | **10** |
| **End term viva** | **20** |

**Annexure 1**

**ANALYSIS AND DESIGN OF ALGORITHMS**

**(CSL 230)**

**Lab Practical Report**



Faculty name:

Sandeep Sir Student name: Manish Kumar Sharma

Roll No.: 19csu173

Semester: 4

Group: FS A1

**Department of Computer Science and Engineering**

**NorthCap University, Gurugram- 122001, India**

**Session 2019-20**

**INDEX**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **S.No** | **Experiment** | **Page No.** | **Date of Experiment** | **Date of Submission** | **Marks** | **CO Covered** | **Sign** |
| **1** | Implement and Calculate the time and space complexity of following programs:  I)Factorial of a number  II)Fibonacci series |  |  |  |  |  |  |
| **2** | Implement Insertion & Selection sort algorithm and compute its time and space complexities. |  |  |  |  |  |  |
| **3** | Implement Merge sort & Quick sort algorithm. Compute its time and space complexities |  |  |  |  |  |  |
| **4** | Implement Strassen’s matrix algorithm. Compute its time complexity. |  |  |  |  |  |  |
| **5** | Implement fractional knapsack algorithm. |  |  |  |  |  |  |
| **6** | Implement Prim’s and Kruskal’s algorithms. Compute and compare their space and time complexities. |  |  |  |  |  |  |
| **7** | Implement Bellman Ford’s Algorithm |  |  |  |  |  |  |
| **8** | Implement 0/1 Knapsack algorithm using Dynamic programming |  |  |  |  |  |  |
| **9** | Implement travelling salesman algorithm using Dynamic Programming |  |  |  |  |  |  |
| **10** | Implement n- queens problem using backtracking |  |  |  |  |  |  |

**EXPERIMENT NO. 1**

|  |
| --- |
| **Student Name and Roll Number: Manish Kumar Sharma and 19csu173** |
| **Semester /Section: 4th / FS A1** |
| **Link to Code:** [**https://github.com/Manish123Sharma/ADA/blob/main/Factorial.java**](https://github.com/Manish123Sharma/ADA/blob/main/Factorial.java)  [**https://github.com/Manish123Sharma/ADA/blob/main/Fibonacci.java**](https://github.com/Manish123Sharma/ADA/blob/main/Fibonacci.java) |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  The students will be able to design and perform analysis of algorithms for factorial of a number and Fibonacci number by applying different notations. |
| **Outcome:**  After completion of lab, students will be able to compare complexities of iterative versus recursive algorithm. |
| **Problem Statement:**   * 1. Write a program for factorial of a number (Iterative/ Recursive) and calculate the time and space complexity of the program.   2. Write a program for Fibonacci series (Recursive) and calculate the time and space complexity of the program |
| **Background Study:**  **Analysis of Fibonacci algorithm**  The factorial algorithm can be expressed using simple recursion:  fun fact 0 = 1  fact n = n \* fact (n - 1);  Looking at the computation that has to be done, we might identify three things that will consume time: integer multiplication, integer subtraction and recursive function calls. Let us ignore the cost of making recursive calls, and suppose that the cost of a multiplication is M and that that of a subtraction is S. We can then define a function T(*n*), meaning “the time required by the algorithm to compute *n*!”, in a very similar form to that of the actual algorithm:  T (0 ) = 0, T(n ) = M + S + T (n - 1) for n > 0  From this,  T (n) = (M + S) + T(n - 1) = 2 (M + S) + T(n - 2 ) = ... = n (M + S) + T (n - n) = n (M + S)  If we regard M and S as being constants, this expression indicates that the time to compute *n*! is proportional to *n* so, for example, computing (2*n*)! will take twice as long as computing *n*! does. A more accurate expression for T(n) would be:  T (n) = (n - 1)lg(n - 1)M + S + T (n - 1)  **Analysis of Fibonacci algorithm:**  In Fibonacci algorithm, there are two conspicuous things that consume time: integer addition and recursive function calls. Letting *A*be a constant representing the time required for a simple addition, we can write down a function *T*(*n*) meaning “the time required by the algorithm to compute the *n*-th Fibonacci number”:  T(0)  =   0  T(1)  =   0  T (n)  =   A + T (n - 2) + T(n - 1)  for n > 1  Using appropriate solution techniques, we discover (to our slight horror) that *T*(*n*) is roughly proportional to 2*n*. This is a very fast rate of growth, and helps to explain why our SML implementations run very slowly, even for modest values of *n*. The problem is the pair of recursive calls, which duplicate much work. A little thought leads to the alternative fastfib algorithm that eliminates one of the recursions, and so has:  T(n) = A  + T(n - 1 )  which is of similar style to the earlier factorial time analysis. Thus, the time requirement is proportional to *n*— a dramatic improvement. Again, as with the factorial algorithm, we might worry about whether the addition operations can be done simply. |

**Student Work Area**

**Algorithm/Flowchart/Code/Sample Outputs**

* + - 1. **Fibonacci Series**

**Algorithm:-**

* **Start**
* **Declare variables i, a,b , show**
* **Initialize the variables, a=0, b=1, and show =0**
* **Enter the number of terms of Fibonacci series to be printed**
* **Print First two terms of series**
* **Loop**

**show=a+b**

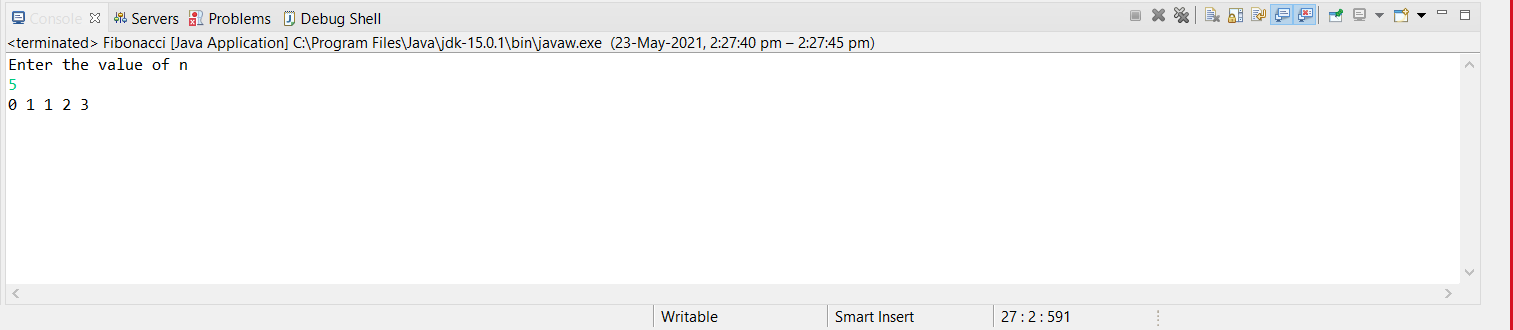
**a=b**

**b=show**

**increase value of i each time by 1**

* **Print show**
* **End**
  + - 1. **Factorial**
  + **Start**
  + **Declare variable num , fact=1 and i=1**
  + **Read num from user**
  + **Repeat till i <n**
  + **Fact = fact \* i and i++**
  + **Print ans fact**
  + **stop**
    - * 1. **Fibonacci**

1. **package** class\_ques;
2. **import** java.util.Scanner;
3. **public** **class** Fibonacci
4. {
5. **static** **int** *a*=0,*b*=1,*c*=0;
6. **static** **void** printFibonacci(**int** n)
7. {
8. **if**(n>0)
9. {
10. *c* = *a* + *b*;
11. *a* = *b*;
12. *b* = *c*;
13. System.***out***.print(" "+*c*);
14. *printFibonacci*(n-1);
15. }
16. }
17. **public** **static** **void** main(String args[])
18. {
19. System.***out***.println("Enter the value of n");
20. Scanner scan = **new** Scanner(System.***in***);
21. **int** n= scan.nextInt();
22. System.***out***.print(*a*+" "+*b*);
23. *printFibonacci*(n-2);
24. }
25. }

****

**b. Factorial**

**package** class\_ques;

**import** java.util.Scanner;

**public** **class** Factorial

{

**static** **int** fact(**int** n){

**if** (n == 0)

**return** 1;

**else**

**return**(n \* *fact*(n-1));

}

**public** **static** **void** main(String args[]){

**int** i,fact=1;

System.***out***.println("Enter the Value of n");

Scanner scan = **new** Scanner(System.***in***);

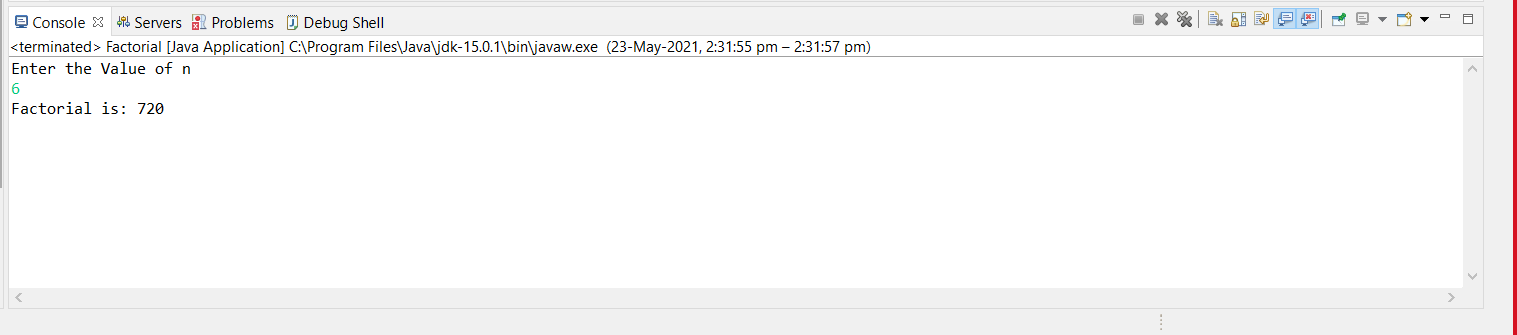
**int** num= scan.nextInt();

**int** f = *fact*(num);

System.***out***.print("Factorial is: "+f);

}

}

****

**EXPERIMENT NO. 2**

|  |
| --- |
| **Student Name and Roll Number: Manish Kumar Sharma and 19csu173** |
| **Semester /Section: 4th / FS A1** |
| **Link to Code:** [**https://github.com/Manish123Sharma/ADA/blob/main/SelectionSort.java**](https://github.com/Manish123Sharma/ADA/blob/main/SelectionSort.java)  [**https://github.com/Manish123Sharma/ADA/blob/main/Insertion.java**](https://github.com/Manish123Sharma/ADA/blob/main/Insertion.java) |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective:**  Perform insertion sort algorithm and compute its time and space complexities. |
| **Outcome:**  Students will be able to calculate execution time and space complexity of the exchange sort algorithms. |
| **Problem Statement:**   * 1. Write a program to implement insertion sort algorithm.   2. Write a program to implement selection sort algorithm. |
| **Background Study:**  **Analysis of the Insertion Sort:**  To insert the last element, we need at most N-1 comparisons and N-1 movements. To insert the N-1st element we need N-2 comparisons and N-2 movements. To insert the 2nd element, we need 1 comparison and one movement.  To sum up:  2\* (1 + 2 + 3 +… N - 1) = 2 \* (N - 1) \* N / 2 = (N-1) \* N = Θ (N2)  If the greater part of the array is sorted, the complexity is almost O(N). The average complexity is proved to be = Θ (N2).  **Analysis of the Selection Sort:**   * 1st iteration of outer loop: inner executes N - 1 times * 2nd iteration of outer loop: inner executes N - 2 times * ... * Nth iteration of outer loop: inner executes 0 times   This is our old favorite sum:  N-1 + N-2 + ... + 3 + 2 + 1 + 0  Which we know is O (N2). |

**Student Work Area**

**Algorithm/Flowchart/Code/Sample Outputs**

**Algorithm:-**

1. **Selection Sort**

**It finds the first smallest element.**

**It swaps it with the first element of the unordered list.**

**It finds the second smallest element.**

**It swaps it with the second element of the unordered list.**

**Similarly, it continues to sort the given elements.**

1. **Insertion Sort**

**If first element is already sorted then return 1;**

**Pick next element**

**Compare with all elements in the sorted sublist**

**Shift all the elements in the sorted sublist that is greater than the value to be sorted**

**Insert the value at that position**

**Repeat until list is sorted**

1. **Selection Sort**

**package** class\_ques;

**import** java.util.Scanner;

**public** **class** SelectionSort {

**static** **int**[] sSort(**int** arr[])

{

**int** i,j,min=0,temp=0;

**for**(i=0;i<4;i++)

{

min=i;

**for**(j=i+1;j<5;j++)

{

**if**(arr[j]<arr[min])

{

min = j;

}

}

temp = arr[min];

arr[min] = arr[i];

arr[i] = temp;

}

**return** arr;

}

**public** **static** **void** main(String args[])

{

Scanner sc = **new** Scanner(System.***in***);

**int** i;

**int** arr[] = **new** **int**[5];

**for**(i=0;i<5;i++)

{

System.***out***.print("enter the value == ");

arr[i] = sc.nextInt();

}

**int** a[] = **new** **int**[5];

a= *sSort*(arr);

System.***out***.println("SELECTION SORT= ");

**for**(i=0;i<5;i++)

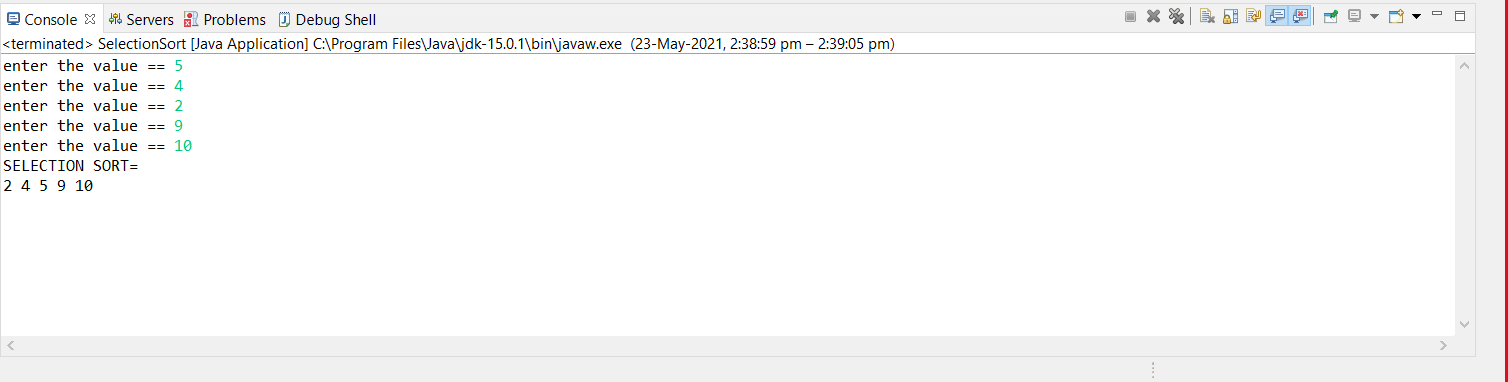
{

System.***out***.print(a[i]+ " ");

}

}

}

****

1. **Insertion Sort**

**package** class\_ques;

**import** java.util.Scanner;

**public** **class** Insertion

{

**static** **int**[] iSort(**int** arr[])

{

**int** i,j,k;

**for**(i=1;i<5;i++) {

k = arr[i];

**for** (j=i-1;j>=0 && arr[j]>k;j--)

{

arr[j+1]=arr[j];

}

arr[j+1] = k;

}

**return** arr;

}

**public** **static** **void** main(String args[]) {

Scanner sc = **new** Scanner(System.***in***);

**int** i;

**int** arr[] = **new** **int**[5];

**for**(i=0;i<5;i++) {

System.***out***.print("enter the value == ");

arr[i] = sc.nextInt();

}

**int** a[] = **new** **int**[5];

a= *iSort*(arr);

System.***out***.println("INSERTION SORT= ");

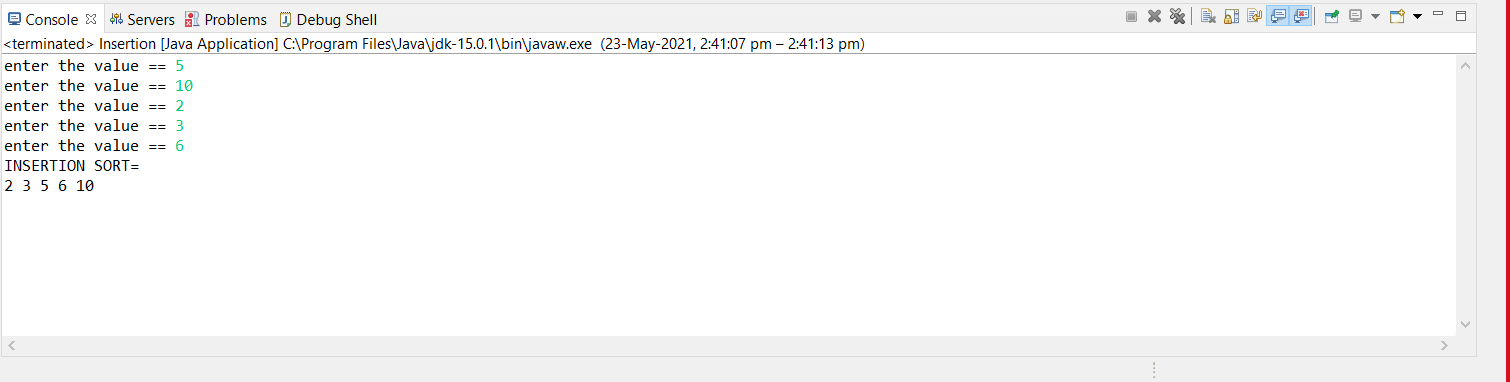
**for**(i=0;i<5;i++) {

System.***out***.print(a[i]+ " ");

}

}

}

****

**EXPERIMENT NO. 3**

|  |
| --- |
| **Student Name and Roll Number: Manish Kumar Sharma and 19csu173** |
| **Semester /Section: 4th /FS A1** |
| **Link to Code:** [**https://github.com/Manish123Sharma/ADA/blob/main/Merge\_sort.java**](https://github.com/Manish123Sharma/ADA/blob/main/Merge_sort.java)  [**https://github.com/Manish123Sharma/ADA/blob/main/Quick\_sort.java**](https://github.com/Manish123Sharma/ADA/blob/main/Quick_sort.java) |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  Perform problems based on divide and conquer technique. |
| **Outcome:**  Students will be able to understand the concept of divide and conquer algorithmic technique. |
| **Problem Statement:**  3.1 Write a program to implement merge sort algorithm.  3.2 Write a program to implement quick sort algorithm. |
| **Background Study:**  **Complexity Analysis:**  The straightforward version of function merge requires at most 2n steps (n steps for copying the sequence to the intermediate array *b*, and at most n steps for copying it back to array *a*). The [time complexity](http://www.inf.fh-flensburg.de/lang/algorithmen/asympen.htm) of merge sort is therefore  T(n) <= 2n + 2 T(n/2)   and  T(1)  =  0  The solution of this recursion yields  T(n) <= 2n log(n)  element  O(n log(n))  Thus, the mergesort algorithm is optimal, since the [lower bound](http://www.inf.fh-flensburg.de/lang/algorithmen/asympen.htm) for the sorting problem of Ω(n log(n)) is attained. In the more efficient variant, function merge requires at most 1.5n steps (n/2 steps for copying the first half of the sequence to the intermediate array b, n/2 steps for copying it back to array *a*, and at most *n*/2 steps for processing the second half). This yields a running time of mergesort of at most 1.5 *n* log (*n*) steps. Algorithm mergesort has a time complexity of Θ(*n* log(*n*)) which is optimal. A drawback of mergesort is that it needs an additional space of Θ(*n*) for the temporary array *b*. There are different possibilities to implement function merge. The most efficient of these is variant *b*. It requires only half as much additional space, it is faster than the other variants, and it is stable. |

**Student Work Area**

**Algorithm/Flowchart/Code/Sample Outputs**

**Algorithm:-**

1. **Merge Sort**

**Start**

**Declare an array of length N**

**Check if N=1, Array is already sorted**

**If N is greater than 1,**

**• set l = 0, r = N-1**

**• compute middle = (l + r)/2**

**• Call mergeSort (Array,l,m) 🡪 first half of the array**

**• Call mergeSort (Array,m+1,r) 🡪 second half of the array**

**• Call merge (Array, l, m, r) to merge arrays sorted**

**End**

1. **Quick Sort**

**Start**

**Quicksort (array, l , h)**

**Declare array to be sorted**

**L = 1st element and h = last element and pivot**

**If(l<h)**

**Pivot = partition (array , l ,h)**

**Quicksort(array, l ,pivot-1)**

**Quicksort(array , pivot+1 , high)**

**End**

* + - 1. **Quick Sort**

**package** class\_ques;

**public** **class** Quick\_sort

{

**static** **int** *b* =0;

**int** partition(**int** arr[],**int** lb ,**int** ub)

{

**int** beg=lb;

**int** end = ub;

**int** key=arr[beg];

**while**(beg<end)

{

**while**(arr[beg]<=key && beg<ub)

{

beg++;

}

**while**(arr[end]>key && end>lb)

{

end--;

}

**if**(beg<end)

{

**int** n =arr[end];

arr[end]=arr[beg];

arr[beg]=n;

}

}

**int** n =arr[lb];

arr[lb]=arr[end];

arr[end]=n;

**return** end;

}

**void** Sort(**int** arr[], **int** lb, **int** ub)

{

**if**(lb<ub)

{

**int** a = partition(arr,lb,ub);

Sort(arr,lb,a-1);

Sort(arr,a+1,ub);

}

*b*++;

}

**static** **void** printArray(**int** arr[])

{

**int** n = arr.length;

**for**(**int** i = 0; i < n; ++i)

{

System.***out***.print(arr[i] + " ");

}

System.***out***.println();

}

**public** **static** **void** main(String args[])

{

Quick\_sort qs2 = **new** Quick\_sort();

**int** arr[] = {8,9,8,3,2,5,6,1,6,7};

**int** arr1[]= {1,2,3,4,5,6,7,8,9,10};

**int** arr2[]= {6,6,6,6,6,6,6,6,6,6};

**int** n = arr.length;

qs2.Sort(arr,0, n-1);

*printArray*(arr);

System.***out***.println(" Recursive Calls --> " +*b*);

*b*=0;

qs2.Sort(arr1,0, n-1);

*printArray*(arr1);

System.***out***.println(" Recursive Calls --> " +*b*);

*b*=0;

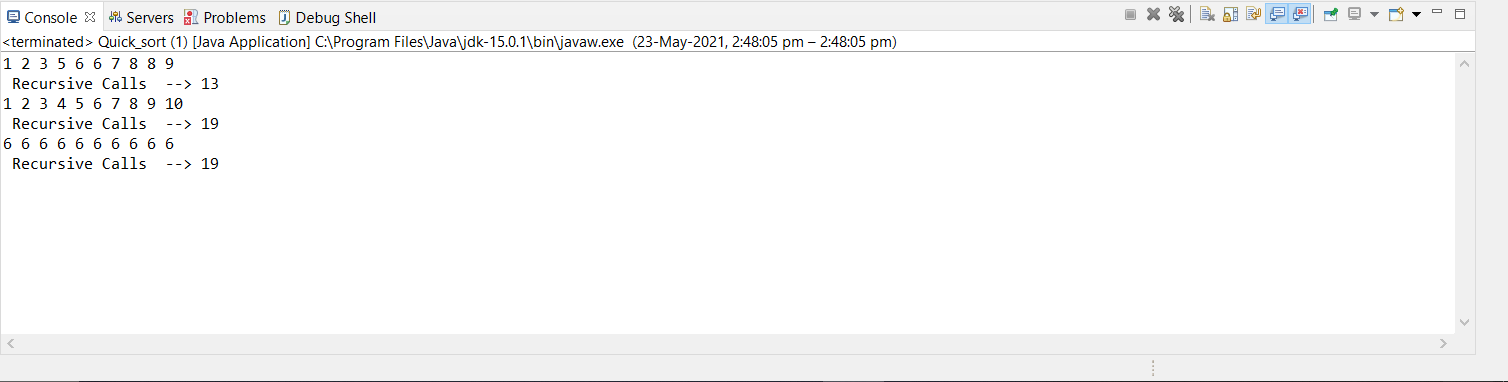
qs2.Sort(arr2,0, n-1);

*printArray*(arr2);

System.***out***.println(" Recursive Calls --> " +*b*);

}

}

****

* + - 1. **Merge Sort**

**package** class\_ques;

**import** java.util.Random;

**public** **class** Merge\_sort

{

**public** **static** **void** main(String[] args)

{

Random random = **new** Random();

**int**[] randomNumbers = **new** **int**[100];

**int**[] ascendingOrderNumbers = **new** **int**[100];

**int**[] descendingOrderNumbers = **new** **int**[100];

**int**[] SameNumbers = **new** **int**[100];

//Random Numbers

**for**(**int** i=0;i<100;i++)

{

**int** randomNumber = random.nextInt(1000);

randomNumbers[i] = randomNumber;

}

// Ascending Order Numbers

**for**(**int** i = 0; i < 100; i++)

{

ascendingOrderNumbers[i] = i;

}

// Descending Order Numbers

**for**(**int** i = 100; i > 0; i--)

{

descendingOrderNumbers[100 - i] = i;

}

**for**(**int** i = 0; i < 100; i++)

{

// int randomNumber = random.nextInt(1000);

SameNumbers[i] = 7;

}

// New Instance for the Solution Class

Merge\_sort solution = **new** Merge\_sort();

// Time Calculation for Random Numbers

**long** startRandom = System.*nanoTime*();

// Merge Sort for Random Numbers

solution.sort(randomNumbers,0,randomNumbers.length-1);

**long** endRandom = System.*nanoTime*();

**long** timeRandom = endRandom - startRandom;

// Time Calculation for Ascending Numbers

**long** startAscending = System.*nanoTime*();

// Merge Sort for Ascending Numbers

solution.sort(ascendingOrderNumbers,0,ascendingOrderNumbers.length-1);

**long** endAscending = System.*nanoTime*();

**long** timeAscending = endAscending - startAscending;

// Time Calculation for Descending Numbers

**long** startDescending = System.*nanoTime*();

// Merge Sort for Descending Numbers

solution.sort(descendingOrderNumbers,0,descendingOrderNumbers.length-1);

**long** endDescending = System.*nanoTime*();

**long** timeDescending = endDescending - startDescending;

**long** startSame = System.*nanoTime*();

// Merge Sort for Random Numbers

solution.sort(randomNumbers,0,randomNumbers.length-1);

**long** endSame = System.*nanoTime*();

**long** timeSame = endSame - startSame;

*printArray*(randomNumbers);

*printArray*(ascendingOrderNumbers);

*printArray*(descendingOrderNumbers);

*printArray*(SameNumbers);

System.***out***.println("Time for merge Sort for Random Numbers : " +timeRandom);

System.***out***.println("Time for merge Sort for Ascending Numbers : " +timeAscending);

System.***out***.println("Time for merge Sort for Descending Numbers : " +timeDescending);

System.***out***.println("Time for merge Sort for Same Numbers : " +timeSame);

}

**void** sort(**int** arr[], **int** beg, **int** end)

{

**if**(beg<end)

{

**int** mid = (beg+end)/2;

sort(arr, beg, mid);

sort(arr , mid+1, end);

merge(arr, beg, mid, end);

}

}

**void** merge(**int** arr[], **int** beg, **int** mid, **int** end)

{

**int** l = mid - beg + 1;

**int** r = end - mid;

**int** LeftArray[] = **new** **int** [l];

**int** RightArray[] = **new** **int** [r];

**for**(**int** i=0; i<l; ++i)

{

LeftArray[i] = arr[beg + i];

}

**for**(**int** j=0; j<r; ++j)

{

RightArray[j] = arr[mid + 1+ j];

}

**int** i = 0, j = 0;

**int** k = beg;

**while**(i<l&&j<r)

{

**if**(LeftArray[i] <= RightArray[j])

{

arr[k] = LeftArray[i];

i++;

}

**else**

{

arr[k] = RightArray[j];

j++;

}

k++;

}

**while**(i<l)

{

arr[k] = LeftArray[i];

i++;

k++;

}

**while**(j<r)

{

arr[k] = RightArray[j];

j++;

k++;

}

}

**static** **void** printArray(**int** arr[])

{

**int** n = arr.length;

**for** (**int** i = 0; i < n; ++i)

{

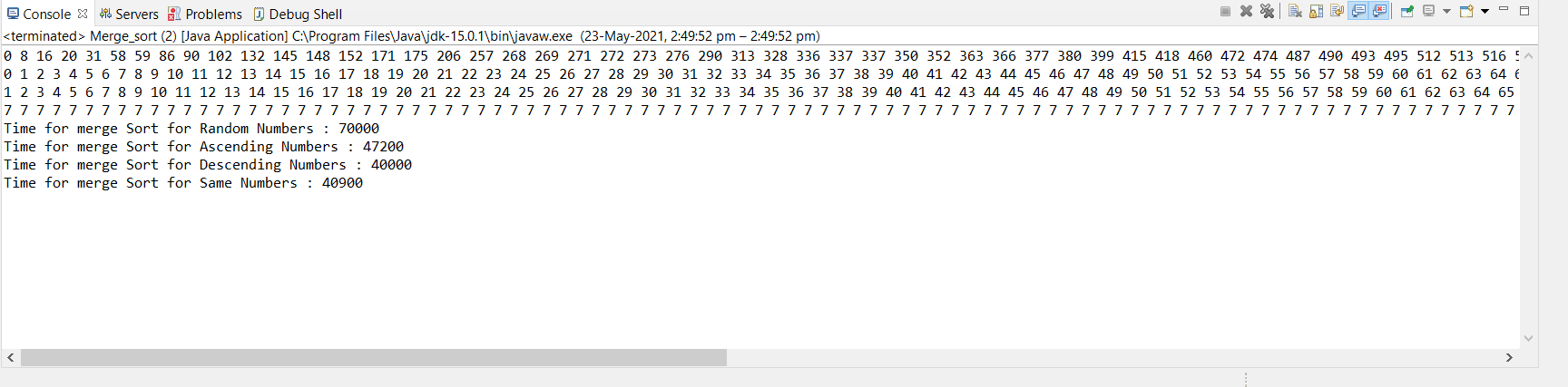
System.***out***.print(arr[i] + " ");

}

System.***out***.println();

}

}

****

**EXPERIMENT NO. 4**

|  |
| --- |
| **Student Name and Roll Number: Manish Kumar Sharma and 19csu173** |
| **Semester /Section: 4th / FS A1** |
| **Link to Code:** [**https://github.com/Manish123Sharma/ADA/blob/main/Strassen.java**](https://github.com/Manish123Sharma/ADA/blob/main/Strassen.java) |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective:**  Perform problems based on divide and conquer technique. |
| **Outcome:**  Students will be able to understand the concept of divide and conquer algorithmic technique and apply on real world problems. |
| **Problem Statement:**  Write a program to implement Strassen’s matrix algorithm. Compute its time complexity. |
| **Background Study:**  **Complexity Analysis**  Applying Master’s method, we get |

**Student Work Area**

**Algorithm/Flowchart/Code/Sample Outputs**

**package** class\_ques;

**public** **class** Strassen

{

**public** **int**[][] multiply(**int**[][] A, **int**[][] B)

{

**int** n = A.length;

**int**[][] R = **new** **int**[n][n];

**if** (n == 1)

R[0][0] = A[0][0] \* B[0][0];

**else** {

**int**[][] A11 = **new** **int**[n / 2][n / 2];

**int**[][] A12 = **new** **int**[n / 2][n / 2];

**int**[][] A21 = **new** **int**[n / 2][n / 2];

**int**[][] A22 = **new** **int**[n / 2][n / 2];

**int**[][] B11 = **new** **int**[n / 2][n / 2];

**int**[][] B12 = **new** **int**[n / 2][n / 2];

**int**[][] B21 = **new** **int**[n / 2][n / 2];

**int**[][] B22 = **new** **int**[n / 2][n / 2];

split(A, A11, 0, 0);

split(A, A12, 0, n / 2);

split(A, A21, n / 2, 0);

split(A, A22, n / 2, n / 2);

split(B, B11, 0, 0);

split(B, B12, 0, n / 2);

split(B, B21, n / 2, 0);

split(B, B22, n / 2, n / 2);

**int**[][] M1

= multiply(add(A11, A22), add(B11, B22));

**int**[][] M2 = multiply(add(A21, A22), B11);

**int**[][] M3 = multiply(A11, sub(B12, B22));

**int**[][] M4 = multiply(A22, sub(B21, B11));

**int**[][] M5 = multiply(add(A11, A12), B22);

**int**[][] M6

= multiply(sub(A21, A11), add(B11, B12));

**int**[][] M7

= multiply(sub(A12, A22), add(B21, B22));

**int**[][] C11 = add(sub(add(M1, M4), M5), M7);

**int**[][] C12 = add(M3, M5);

**int**[][] C21 = add(M2, M4);

**int**[][] C22 = add(sub(add(M1, M3), M2), M6);

join(C11, R, 0, 0);

join(C12, R, 0, n / 2);

join(C21, R, n / 2, 0);

join(C22, R, n / 2, n / 2);

}

**return** R;

}

**public** **int**[][] sub(**int**[][] A, **int**[][] B)

{

**int** n = A.length;

**int**[][] C = **new** **int**[n][n];

**for** (**int** i = 0; i < n; i++)

**for** (**int** j = 0; j < n; j++)

C[i][j] = A[i][j] - B[i][j];

**return** C;

}

**public** **int**[][] add(**int**[][] A, **int**[][] B)

{

**int** n = A.length;

**int**[][] C = **new** **int**[n][n];

**for** (**int** i = 0; i < n; i++)

**for** (**int** j = 0; j < n; j++)

C[i][j] = A[i][j] + B[i][j];

**return** C;

}

**public** **void** split(**int**[][] P, **int**[][] C, **int** iB, **int** jB)

{

**for** (**int** i1 = 0, i2 = iB; i1 < C.length; i1++, i2++)

**for** (**int** j1 = 0, j2 = jB; j1 < C.length;

j1++, j2++)

C[i1][j1] = P[i2][j2];

}

**public** **void** join(**int**[][] C, **int**[][] P, **int** iB, **int** jB)

{

**for** (**int** i1 = 0, i2 = iB; i1 < C.length; i1++, i2++)

**for** (**int** j1 = 0, j2 = jB; j1 < C.length;

j1++, j2++)

P[i2][j2] = C[i1][j1];

}

**public** **static** **void** main(String[] args)

{

Strassen s = **new** Strassen();

**int** N = 4;

**int**[][] A = { { 1, 2, 3, 4 },

{ 4, 3, 0, 1 },

{ 5, 6, 1, 1 },

{ 0, 2, 5, 6 } };

**int**[][] B = { { 1, 0, 5, 1 },

{ 1, 2, 0, 2 },

{ 0, 3, 2, 3 },

{ 1, 2, 1, 2 } };

**int**[][] C = s.multiply(A, B);

System.***out***.println();

**for** (**int** i = 0; i < N; i++) {

**for** (**int** j = 0; j < N; j++)

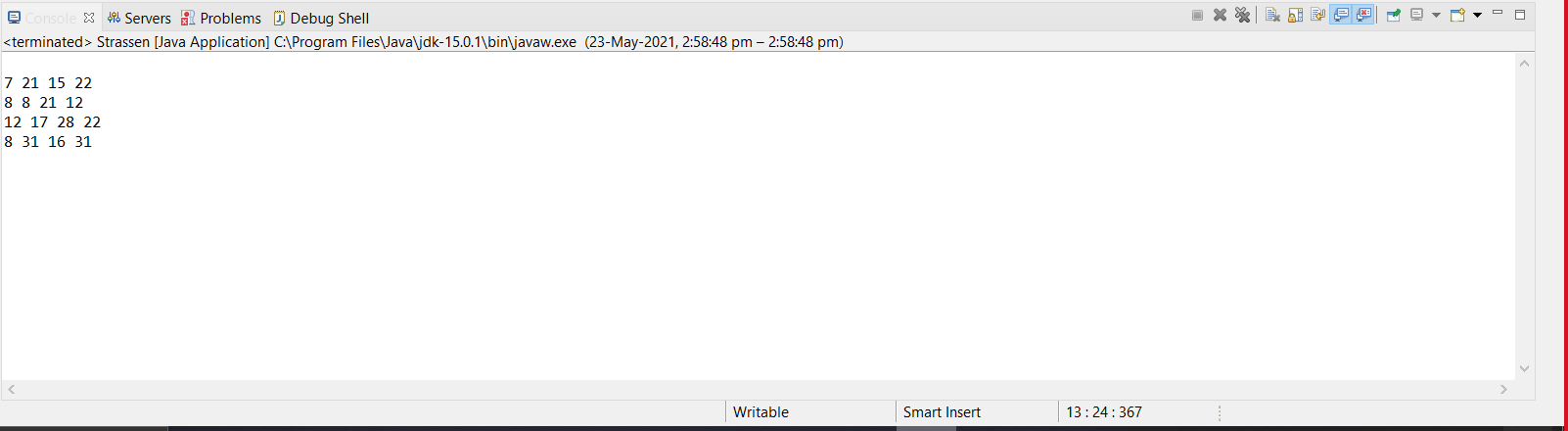
System.***out***.print(C[i][j] + " ");

System.***out***.println();

}

}

}

****

**EXPERIMENT NO. 5**

|  |
| --- |
| **Student Name and Roll Number: Manish Kumar Sharma and 19csu173** |
| **Semester /Section:4th /FS A1** |
| **Link to Code:** [**https://github.com/Manish123Sharma/ADA/blob/main/Knapsack.java**](https://github.com/Manish123Sharma/ADA/blob/main/Knapsack.java) |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  Perform problems based on Greedy algorithmic technique. |
| **Outcome:**  Students will be able to understand the concept of greedy algorithm. |
| **Problem Statement:**  Implement Fractional Knapsack Algorithm. |
| **Background Study:**  **Complexity Analysis**  If the items are already sorted into decreasing order of vi / wi, then the while-loop takes a time in *O(n)*.  Therefore, the total time including the sort is in *O(n log n)*. If we keep the items in heap with largest vi/wi at the root. Then   * creating the heap takes *O(n)* time * while-loop now takes *O(log n)* time |

**Student Work Area**

**Algorithm/Flowchart/Code/Sample Outputs**

**Algorithm:-**

* **Knapsack (Array W, Array V, int M)**
* **for i 1 to size (V)**

**calculate cost[i] V[i] / W[i]**

* **Sort-Descending (cost)**
* **i ← 1**
* **while (i <= size(V))**

**if W[i] <= M**

**M ← M – W[i]**

**total ← total + V[i];**

**if W[i] > M**

**i ← i+1**

**package** class\_ques;

**import** java.util.\*;

**public** **class** Knapsack

{

**static** **class** ItemValue

{

Double cost;

**double** wt, val, ind;

**public** ItemValue(**int** wt, **int** val, **int** ind)

{

**this**.wt = wt;

**this**.val = val;

**this**.ind = ind;

cost = ((**double**)val / (**double**)wt);

}

}

**private** **static** **double** getMaxValue(**int**[] wt, **int**[] val,**int** capacity)

{

ItemValue iVal[] = **new** ItemValue[wt.length];

**for**(**int** i = 0; i < wt.length; i++)

{

iVal[i] = **new** ItemValue(wt[i], val[i], i);

}

Arrays.*sort*(iVal, **new** Comparator<ItemValue>()

{

@Override

**public** **int** compare(ItemValue o1, ItemValue o2)

{

**return** o2.cost.compareTo(o1.cost);

}

});

**double** totalValue = 0d;

**for**(ItemValue i : iVal)

{

**int** curWt = (**int**)i.wt;

**int** curVal = (**int**)i.val;

**if**(capacity - curWt >= 0)

{

capacity = capacity - curWt;

totalValue = totalValue + curVal;

}

**else**

{

**double** fraction= ((**double**)capacity / (**double**)curWt);

totalValue += (curVal \* fraction);

capacity= (**int**)(capacity - (curWt \* fraction));

**break**;

}

}

**return** totalValue;

}

**public** **static** **void** main(String[] args)

{

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("enter no. of items");

**int** n = sc.nextInt();

Random random = **new** Random();

**int**[] wt = **new** **int**[n];

**int**[] val = **new** **int**[n];

**for**(**int** i = 0; i < n; i++)

{

**int** randomNumber = random.nextInt(100);

wt[i] = randomNumber;

**int** rannum = random.nextInt(100);

val[i] = rannum + 100;

}

System.***out***.println("enter capacity");

**int** capacity = sc.nextInt();

sc.close();

**double** maxValue = *getMaxValue*(wt, val, capacity);

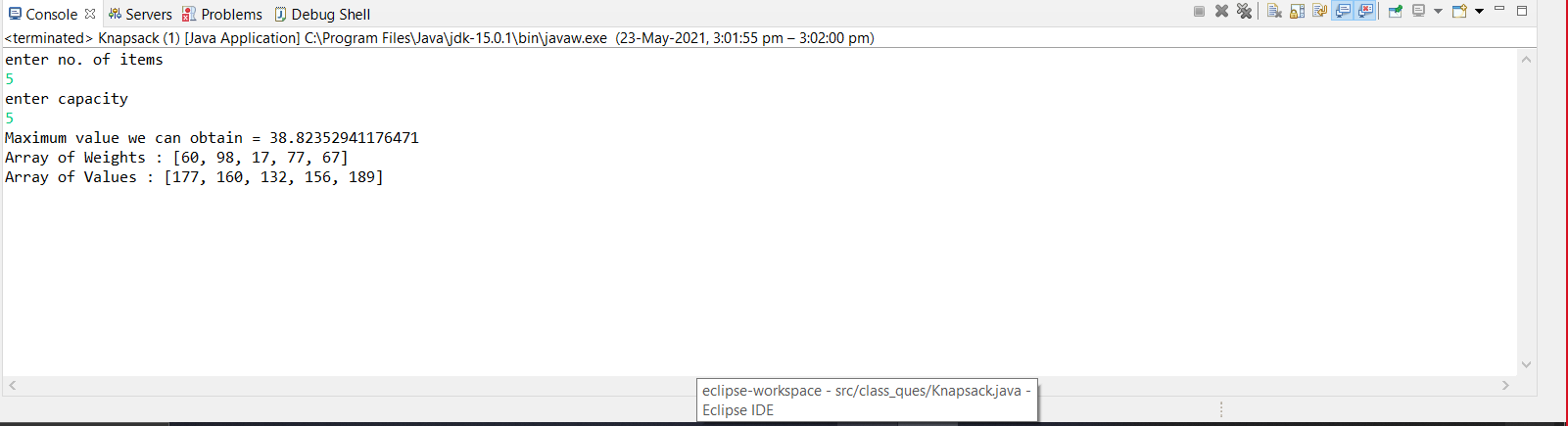
System.***out***.println("Maximum value we can obtain = "+ maxValue);

System.***out***.println("Array of Weights : "+Arrays.*toString*(wt));

System.***out***.println("Array of Values : "+Arrays.*toString*(val));

}

}

****

**EXPERIMENT NO. 6**

|  |
| --- |
| **Student Name and Roll Number: Manish Kumar Sharma and 19csu173** |
| **Semester /Section: 4th / FS A1** |
| **Link to Code:** [**https://github.com/Manish123Sharma/ADA/blob/main/Prims.java**](https://github.com/Manish123Sharma/ADA/blob/main/Prims.java)  [**https://github.com/Manish123Sharma/ADA/blob/main/Kruskals.java**](https://github.com/Manish123Sharma/ADA/blob/main/Kruskals.java) |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective:**  Perform problems based on Greedy algorithmic technique. |
| **Outcome:**  Students will be able to understand the concept of greedy algorithm. |
| **Problem Statement:**  Implement Prim’s and Kruskal’s algorithms for finding minimum spanning tree of a given graph. |
| **Background Study:**  **Complexity Analysis**  **Prim’s Algorithm**  Running Time = O(m + n log n) (m = edges, n = nodes)  If a heap is not used, the run time will be O(n^2) instead of O(m + n log n). However, using a heap complicates the code since you’re complicating the data structure. A Fibonacci heap is the best kind of heap to use, but again, it complicates the code. Unlike Kruskal’s, it doesn’t need to see all of the graph at once. It can deal with it one piece at a time. It also doesn’t need to worry if adding an edge will create a cycle since this algorithm deals primarily with the nodes, and not the edges. For this algorithm the number of nodes needs to be kept to a minimum in addition to the number of edges. For small graphs, the edges matter more, while for large graphs the number of nodes matters more.  **Kruskal’s Algorithm**   * Initialization *O*(*V*) time * Sorting the edges Q(*E* lg *E*) = Q(*E* lg *V*) (why?) * *O*(*E*) calls to FindSet * Union costs   + Let *t(v)* – the number of times *v* is moved to a new cluster   + Each time a vertex is moved to a new cluster the size of the cluster containing the vertex at least doubles: *t(v)* £ log *V*   + Total time spent doing Union   Total time: O(E lg V) |

**Student Work Area**

**Algorithm/Flowchart/Code/Sample Outputs**

1. **Prim’s**

**package** class\_ques;

**public** **class** Prims

{

**public** **static** **void** main(String[] args)

{

**int**[][] graph = { { 100, 1, 4, 2, 100 }, { 3, 100, 100, 5, 7 }, { 100, 8, 100, 100, 7}, { 1, 4, 100, 100, 5 },{ 100, 6, 11, 12, 100 } };

**int** arr[] = *prim*(graph);

**for**(**int** i : arr)

{

System.***out***.println(i);

}

}

**public** **static** **int**[] prim(**int**[][] arr)

{

**int**[] r = { 4, 0, 0, 0, 0 };

**int** rc = 0;

**int** v = 0;

**int** w = 0;

**for**(**int** m = 1; m < 5; m++)

{

**int** min = 100;

**for**(**int** j = 0; j < rc; j++)

{

**for**(**int** i = 0; i < 5; i++)

{

**if**(min > arr[r[j]][i])

{

v = i;

w = r[j];

min = arr[r[j]][i];

}

}

}

rc++;

r[rc] = v;

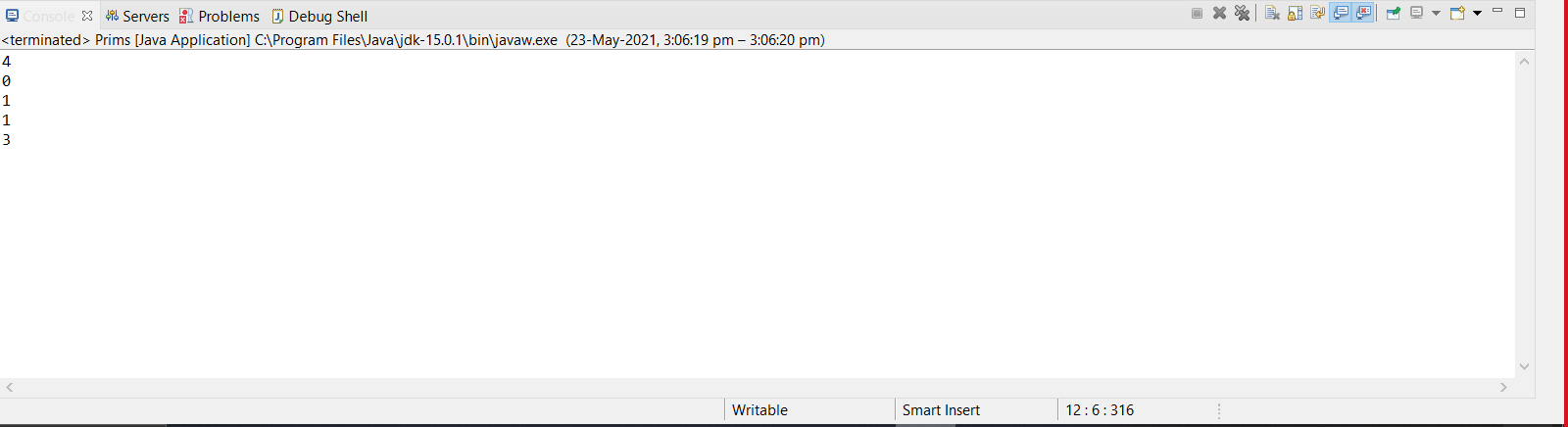
arr[v][w] = arr[w][v] = 100;

}

**return** r;

}

}

****

1. **Kruskal’s**

**package** class\_ques;

**import** java.util.Scanner;

**public** **class** Kruskals

{

**public** **static** **void** main(String[] args)

{

Scanner scan= **new** Scanner (System.***in***);

**int**[][] matrix= **new** **int**[5][5];

**int**[] parent= **new** **int**[5];

**int** min;

**int** u=0;

**int** v=0;

**int** noofedges=1;

**int** total=0;

**for**(**int** i=0;i<5;i++)

{

parent[i]=0;

**for**(**int** j=0;j<5;j++)

{

matrix[i][j]=scan.nextInt();

**if**(matrix[i][j]==0)

{

matrix[i][j]=999;

}

}

}

**while**(noofedges<5)

{

min=999;

**for**(**int** i=0;i<5;i++)

{

**for**(**int** j=0;j<5;j++)

{

**if**(matrix[i][j]<min)

{

min=matrix[i][j];

u=i;

v=j;

}

}

}

**int** x=u;**int** y=v;

**while**(parent[x]!=0)

{

x=parent[x];

}

**while**(parent[y]!=0)

{

y=parent[v];

}

**if**(x!=y)

{

noofedges++;

System.***out***.println("Edge found;"+u+"->"+v+"Min:" +min);

total+=min;

parent[v]=u;

}

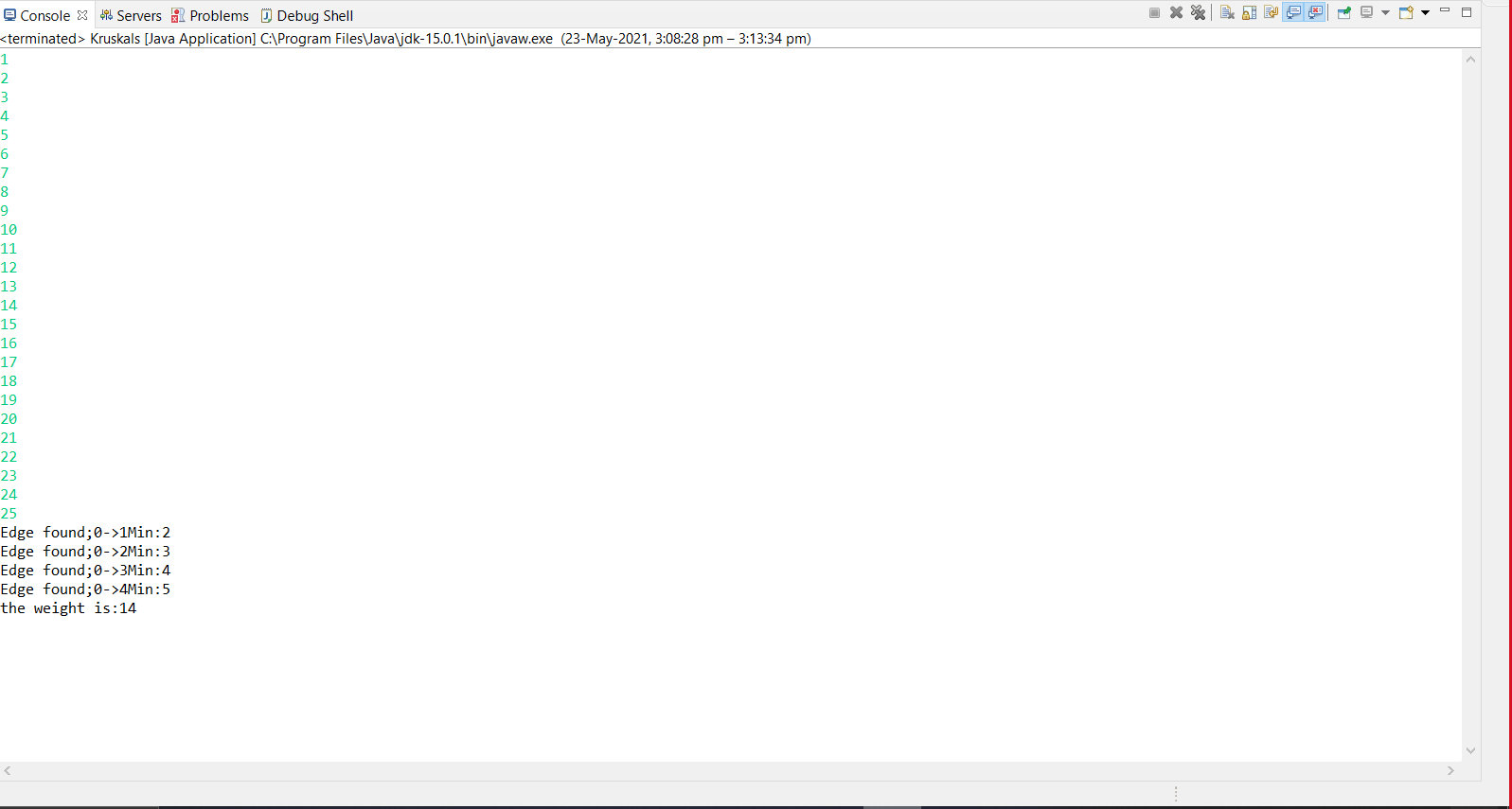
matrix[u][v]=matrix[v][u]=999;

}

System.***out***.println("the weight is:"+total);

}

}

****

**EXPERIMENT NO. 7**

|  |
| --- |
| **Student Name and Roll Number: Manish Kumar Sharma and 19csu173** |
| **Semester /Section: 4th /FS A1** |
| **Link to Code:** [**https://github.com/Manish123Sharma/ADA/blob/main/Bellman\_Ford.java**](https://github.com/Manish123Sharma/ADA/blob/main/Bellman_Ford.java) |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective:**  Implement Bellman Ford’s Algorithm. |
| **Outcome:**  The Bellman-Ford algorithm is a graph search algorithm that finds the shortest path between a given source vertex and all other vertices in the graph. This algorithm can be used on both weighted and unweighted graphs. |
| **Problem Statement:**  Write a program to implement Bellman’s Ford Algorithm. |

**Student Work Area**

**Algorithm/Flowchart/Code/Sample Outputs**

**package** class\_ques;

**public** **class** Bellman\_Ford

{

**public** **static** **void** main(String[] args)

{

**int** V = 5;

**int** E = 8;

**int** graph[][] = {

{ 0, 1, -1 },

{ 0, 2, 6 },

{ 1, 2, 9 },

{ 1, 3, -2 },

{ 1, 4, -7 },

{ 3, 2, 3 },

{ 3, 1, 2 },

{ 4, 3, 8 }

};

*bellmanFordAlgo*(graph, V, E, 0);

}

**public** **static** **void** bellmanFordAlgo(**int** G[][], **int** V, **int** E, **int** source)

{

**int** []dis = **new** **int**[V];

**for** (**int** i = 0; i < V; i++)

{

dis[i] = Integer.***MAX\_VALUE***;

}

dis[source] = 0;

**for** (**int** i = 0; i < V - 1; i++)

{

**for** (**int** j = 0; j < E; j++)

{

**if** (dis[G[j][0]] != Integer.***MAX\_VALUE*** && dis[G[j][0]] + G[j][2] < dis[G[j][1]])dis[G[j][1]] = dis[G[j][0]] + G[j][2];

}

}

**for** (**int** i = 0; i < E; i++)

{

**int** x = G[i][0];

**int** y = G[i][1];

**int** weight = G[i][2];

**if** (dis[x] != Integer.***MAX\_VALUE*** && dis[x] + weight < dis[y])

{

System.***out***.println(" --- Negative Cycle ---");

}

}

System.***out***.println(" Distance of Vertex from Src");

**for** (**int** i = 0; i < V; i++)

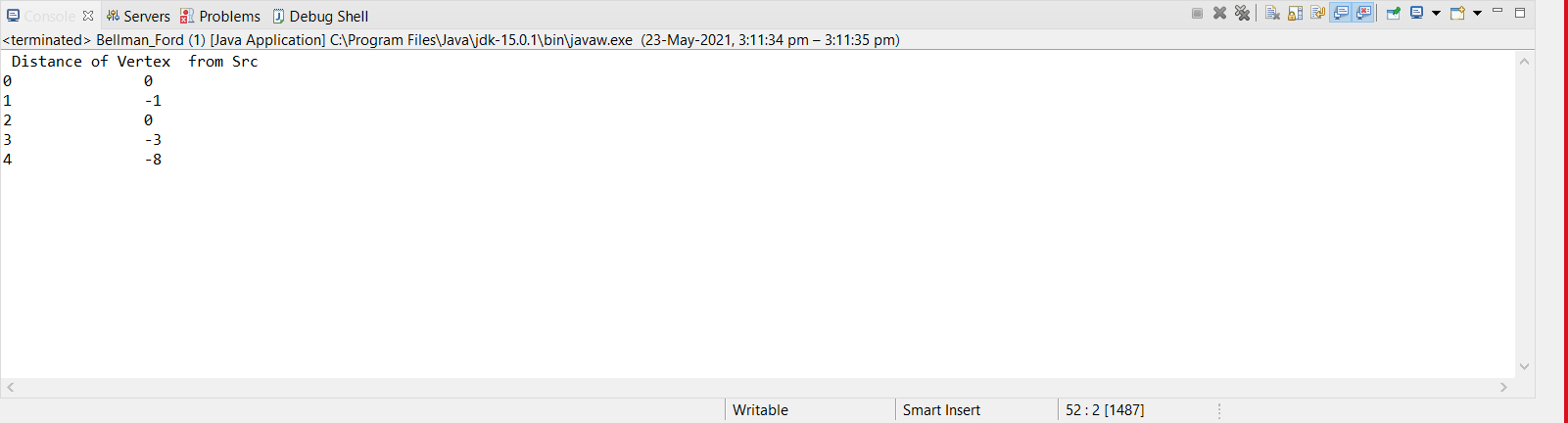
{

System.***out***.println(i + "\t\t" + dis[i]);

}

}

}

****

**EXPERIMENT NO. 8**

|  |
| --- |
| **Student Name and Roll Number: Manish Kumar Sharma and 19csu173** |
| **Semester /Section: 4th / FS A1** |
| **Link to Code:** [**https://github.com/Manish123Sharma/ADA/blob/main/Knapsack01.java**](https://github.com/Manish123Sharma/ADA/blob/main/Knapsack01.java) |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  Perform problems based on Dynamic Programming. |
| **Outcome:**  Students will be able to understand the concept of dynamic programming. |
| **Problem Statement:**  Write a program for 0/1 Knapsack problem using Dynamic Programming. |
| **Background Study:**  This dynamic-0-1-kanpsack algorithm takes θ(*nw*) times, broken up as follows: θ(*nw*) times to fill the *c*-table, which has (*n*+*1*).(*w*+1) entries, each requiring θ(1) time to compute. *O*(*n*) time to trace the solution, because the tracing process starts in row *n* of the table and moves up 1 row at each step. |

**Student Work Area**

**Algorithm/Flowchart/Code/Sample Outputs**

**Algorithm:-**

* **Divide the problem into 2 parts (cases)**
* **Case 1: The item is included in the optimal subset.**
* **Case 2: The item is not included in the optimal set.**
* **Then remove the nth element**

package class\_ques;

import java.util.Random;

import java.util.Scanner;

public class Knapsack01

{

public static int knapsack(int cap, int weight[], int value[], int n)

{

int a[] = new int[cap + 1];

int b[] = new int[cap + 1];

for (int w = 0; w <= cap; w++)

{

b[w] = 0;

}

for (int k = 0; k < n; k++)

{

for (int i = 0; i <= cap; i++)

{

a[i] = b[i];

}

for (int w = weight[k]; w <= cap; w++)

{

if ((a[w - weight[k]] + value[k]) > a[w])

{

// System.out.println(weight[k] + " --> " + a[w - weight[k]]);

b[w] = a[w - weight[k]] + value[k];

}

}

}

return b[cap];

}

public static void main(String[] args)

{

// TODO Auto-generated method stub

Random rn = new Random();

Scanner sc = new Scanner(System.in);

System.out.println("Enter no. of items: ");

int n = sc.nextInt();

int[] weight = new int[n];

for (int i = 0; i < n; i++)

{

weight[i] = rn.nextInt(10);

}

int[] value = new int[n];

for (int i = 0; i < n; i++)

{

value[i] = rn.nextInt(10);

}

System.out.println("Enter capacity: ");

int c = sc.nextInt();

for (int i = 0; i < n; i++)

{

System.out.print(weight[i] + " ");

}

System.out.println();

for (int i = 0; i < n; i++)

{

System.out.print(value[i] + " ");

}

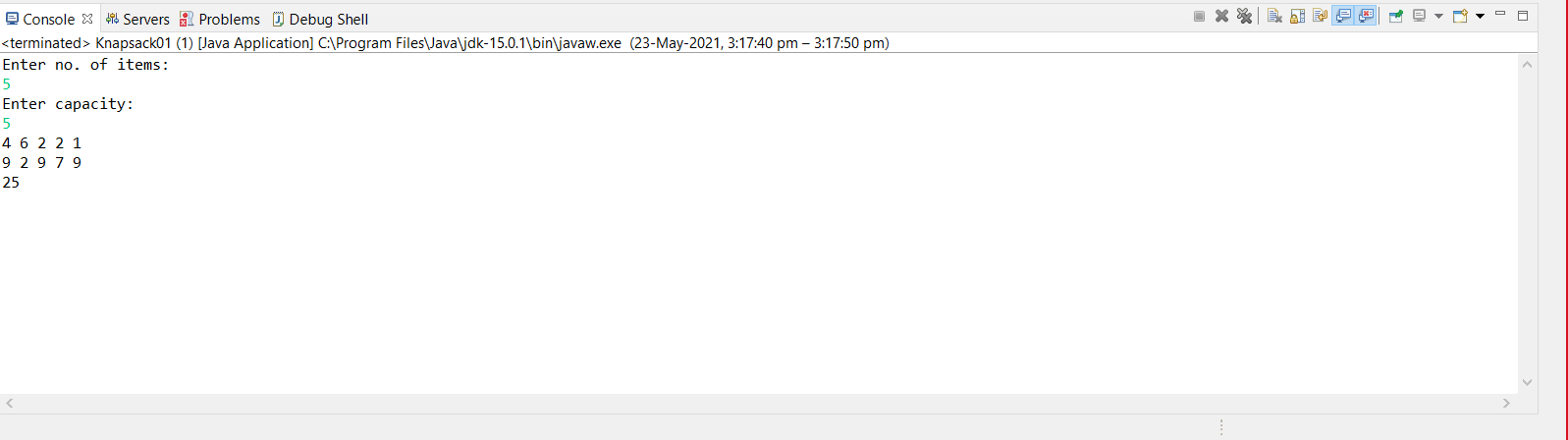
System.out.println();

System.out.println(knapsack(c, weight, value, n));

sc.close();

}

}

****

**EXPERIMENT NO. 9**

|  |
| --- |
| **Student Name and Roll Number: Manish Kumar Sharma and 19csu173** |
| **Semester /Section: 4th / FS A1** |
| **Link to Code:** [**https://github.com/Manish123Sharma/ADA/blob/main/TSP.java**](https://github.com/Manish123Sharma/ADA/blob/main/TSP.java) |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective:**  Implement travelling salesman problem.  **Traveling Salesperson Problem:**  Given *n* cities, a salesperson starts at a specified city (source), visit all n-1 cities only once and return to the city from where he has started. The objective of this problem is to find a route through the cities that minimizes the cost and thereby maximizing the profit. |
| **Outcome:**  Students will be able to understand the concept of dynamic programming. |
| **Problem Statement:**  Write a program for Travelling salesman problem using Dynamic Programming. |
| **Background Study:**  **Complexity Analysis:**  An algorithm that proceeds to find an optimal tour by making use of (1) and (2) will require O(*n*22n) time since the computation of *g*(*i*, *S*) with |*S*| = *k* requires *k* -1 comparisons when solving (2). It’s better than enumerating all *n*! different tours to find the best one. The most serious drawback of this dynamic programming solution is the space needed. The space needed in O(*n*2*n*). This is too large even for modest values of *n*. |

**Student Work Area**

**Algorithm/Flowchart/Code/Sample Outputs**

**Algorithm:-**

* **C ({1}, 1) = 0**
* **for s = 2 to n do**
* **for all subsets S Є {1, 2, 3, … , n} of size s and containing 1**
* **C (S, 1) = ∞**
* **for all j Є S and j ≠ 1**
* **C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}**
* **Return minj C ({1, 2, 3, …, n}, j) + d(j, i)**

**package** class\_ques;

**import** java.util.Scanner;

**public** **class** TSP

{

**static** **int** tsdp(**int** [][] c, **int** [] tour, **int** start, **int** n){

**int** [] mintour = **new** **int**[10], temp = **new** **int**[10];

**int** mincost = 9999, ccost, i, j, k;

**if** (start == n - 1) {

**return** (c[tour[n-1]][tour[n]]+c[tour[n]][1]);

}

**for** (i = start + 1; i <= n; i++) {

**for** (j = 1; j <= n; j++) {

temp[j] = tour[j];

}

}

temp[start + 1] = tour[i];

temp[i] = tour[start + 1];

**if** (c[tour[start]][tour[i]] + (ccost=*tsdp*(c, temp, start+1, n)) < mincost) {

mincost = c[tour[start]][tour[i]] + ccost;

**for** (k = 1; k <=n; k++) {

mintour[k] = temp[k];

}

}

**for** (i = 1; i <= n; i++) {

tour[i] = mintour[i];

}

**return** mincost;

}

**public** **static** **void** main(String[] args) {

Scanner z = **new** Scanner(System.***in***);

**int** [][] c = **new** **int** [10][10];

**int** [] tour = **new** **int**[10];

**int** i, j, cost, n;

System.***out***.print("Enter the no of cities: ");

n = z.nextInt();

**if** (n == 1) {

System.***out***.println("Path is not possible.");

System.*exit*(0);

}

System.***out***.println("Enter the cost matrix:");

**for** (i = 1; i <= n; i++) {

**for** (j = 1; j <= n; j++) {

c[i][j] = z.nextInt();

}

}

**for** (i = 1; i <=n; i++) {

tour[i] = i;

}

cost = *tsdp*(c, tour, 1, n);

System.***out***.println("The optimal tour is:");

**for** (i = 1; i <= n; i++) {

System.***out***.print(tour[i] + ":->");

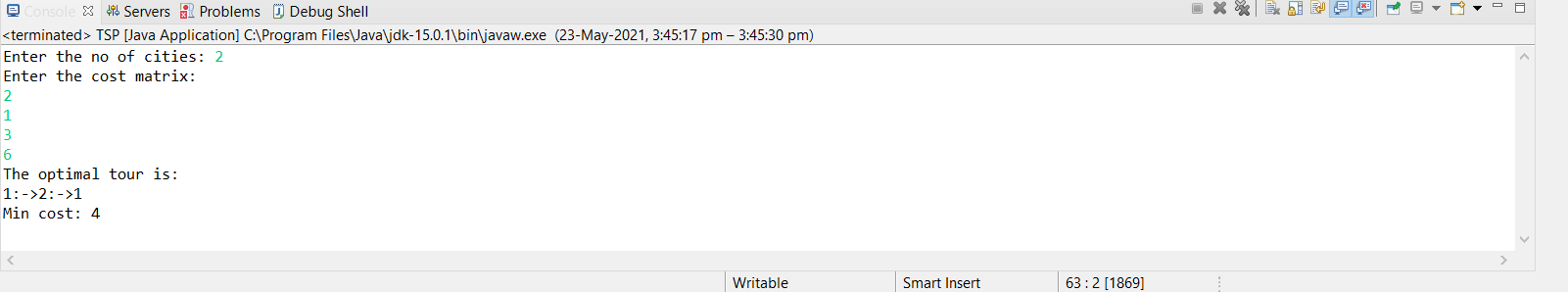
}

System.***out***.println("1");

System.***out***.println("Min cost: " + cost);

}

}

****

**EXPERIMENT NO. 10**

|  |
| --- |
| **Student Name and Roll Number: Manish Kumar Sharma and 19csu173** |
| **Semester /Section: 4th / FS A1** |
| **Link to Code:** <https://github.com/Manish123Sharma/ADA/blob/main/NQueen.java> |
| **Date:** |
| **Faculty Signature:** |
| **Marks:** |

|  |
| --- |
| **Objective(s):**  To Implement n Queen’s Problem |
| **Outcome:**  Students will be able to understand the concept of Backtracking algorithm. |
| **Problem Statement:**  Write a program for N-Queen’s problem using Backtracking. |
| **Background Study:**  **Complexity Analysis:**  The power of the set of all possible solutions of the n queen’s problem is n! and the bounding function takes a linear amount of time to calculate, therefore the running time of the n queens problem is O(n!). |

**Student Work Area**

**Algorithm/Flowchart/Code/Sample Outputs**

**package** class\_ques;

**import** java.util.Scanner;

**public** **class** NQueen

{

**private** **static** **int** *N*;

**private** **static** **int**[][] *board* = **new** **int**[100][100];

**private** **static** **boolean** isAttack(**int** i,**int** j)

{

**int** k,l;

**for**(k=0;k<*N*;k++)

{

**if**((*board*[i][k] == 1) || (*board*[k][j] == 1))

**return** **true**;

}

**for**(k=0;k<*N*;k++)

{

**for**(l=0;l<*N*;l++)

{

**if**(((k+l) == (i+j)) || ((k-l) == (i-j)))

{

**if**(*board*[k][l] == 1)

**return** **true**;

}

}

}

**return** **false**;

}

**private** **static** **boolean** nQueen(**int** n)

{

**int** i,j;

**if**(n==0)

**return** **true**;

**for**(i=0;i<*N*;i++)

{

**for**(j=0;j<*N*;j++)

{

**if**((!*isAttack*(i,j)) && (*board*[i][j]!=1))

{

*board*[i][j] = 1;

**if**(*nQueen*(n-1)==**true**)

{

**return** **true**;

}

*board*[i][j] = 0;

}

}

}

**return** **false**;

}

**public** **static** **void** main(String[] args)

{

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("Enter the value of N for NxN chessboard");

*N* = s.nextInt();

**int** i,j;

*nQueen*(*N*);

**for**(i=0;i<*N*;i++)

{

**for**(j=0;j<*N*;j++)

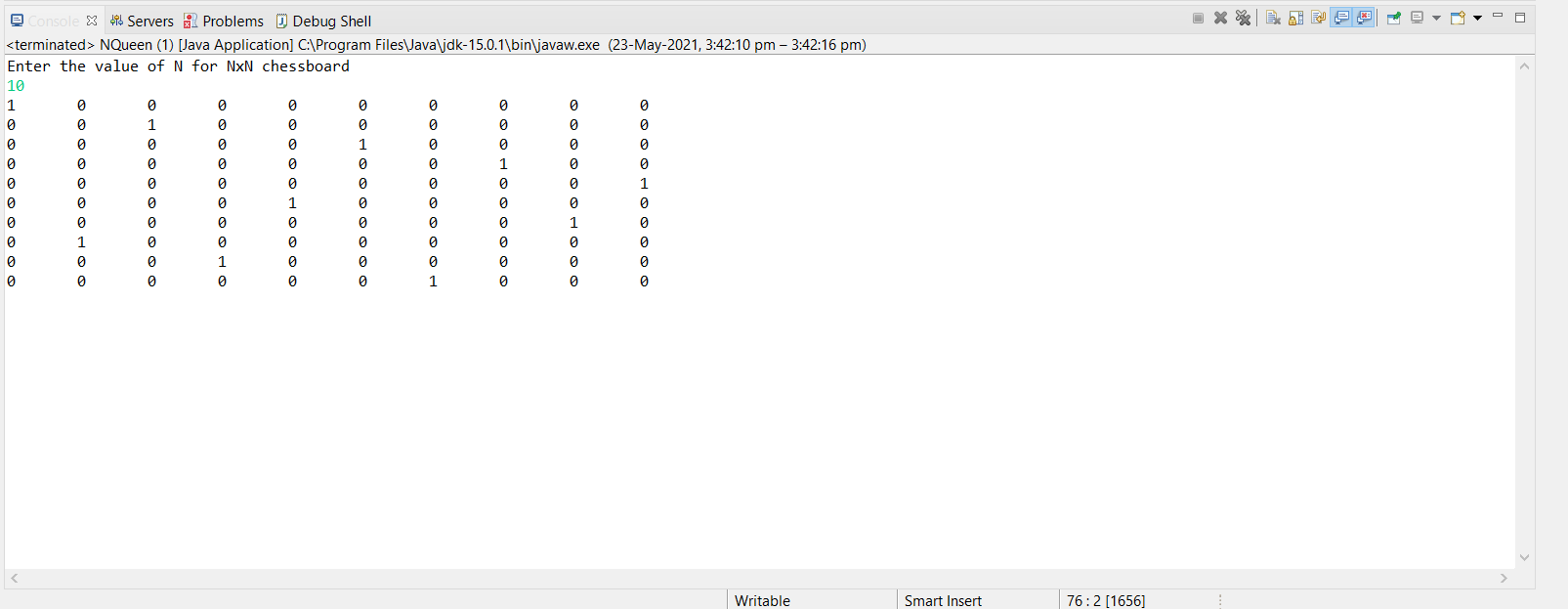
System.***out***.print(*board*[i][j]+"\t");

System.***out***.print("\n");

}

}

}

****