SIR M. VISVESVARAYA INSTITUTE OF TECHNOLOGY

BENGALURU-562157

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

LAB MANUAL

2022-2023

DESIGN & ANALYSIS OF ALGORITHMS LABORATORY COMPONENT

[AS PER CHOICE BASED CREDIT SYSTEM (CBCS) 2021 SCHEME]

21CS42

IV SEMESTER CSE

PREPARED AND COMPILED BY: Dr. SUMA SWAMY

UNDER THE GUIDANCE OF

HOD

Dr. ANITHA T N

Design and Analysis of Algorithms -21CS42- Laboratory Component

DESIGN AND ANALYSIS OF ALGORITHMS- Laboratory Component			
Course Code	**21CS42**	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 T + 20 P	Total Marks	100
Credits	04	Exam Hours	03

Course Learning Objectives:

CLO 1. Explain the methods of analysing the algorithms and to analyze performance of algorithms.
CLO 2. State algorithm's efficiencies using asymptotic notations.
CLO 3. Solve problems using algorithm design methods such as the brute force method, greedy method, divide and conquer, decrease and conquer, transform and conquer, dynamic programming, backtracking and branch and bound.
CLO 4. Choose the appropriate data structure and algorithm design method for a specified application.
CLO 5. Introduce P and NP classes.

Teaching-Learning Process (General Instructions)

These are sample Strategies, which teachers can use to accelerate the attainment of the various course outcomes.

1. Lecturer method (L) does not mean only traditional lecture method, but different type of teaching methods may be adopted to develop the outcomes.
2. Show Video/animation films to explain functioning of various concepts.
3. Encourage collaborative (Group Learning) Learning in the class.
4. Ask at least three HOT (Higher order Thinking) questions in the class, which promotes critical thinking.
5. Adopt Problem Based Learning (PBL), which fosters students' Analytical skills, develop thinking skills such as the ability to evaluate, generalize, and analyze information rather thansimply recall it.
6. Topics will be introduced in a multiple representation.
7. Show the different ways to solve the same problem and encourage the students to come upwith their own creative ways to solve them.
8. Discuss how every concept can be applied to the real world - and when that's possible, it helps improve the students' understanding.

Module-1

Brute force design technique: Selection sort with complexity Analysis.

Laboratory Component:

1. Sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the brute force method works along with its time complexity analysis: worst case, average case and bestcase.

Teaching-Learning Process	1. Problem based Learning. 2. Chalk & board, Active Learning. 3. Laboratory Demonstration.

Design and Analysis of Algorithms -21CS42- Laboratory Component

<table>
<tr><td colspan="2">Module-2</td></tr>
<tr><td colspan="2">Divide and Conquer: Merge sort, Quick sort.</td></tr>
<tr><td colspan="2">Laboratory Component:

1. Sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the divide-and-conquer method works along with its time complexity analysis: worst case, average case and best case.

2. Sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the divide-and-conquer method works along with its time complexity analysis: worst case, average case and best case.</td></tr>
<tr><td>Teaching-Learning Process</td><td>1. Chalk & board, Active Learning, MOOC, Problem based Learning.
2. Laboratory Demonstration.</td></tr>
<tr><td colspan="2">Module-3</td></tr>
<tr><td colspan="2">Greedy Method: Knapsack Problem

Minimum cost spanning trees: Prim’s Algorithm, Kruskal’s Algorithm with performance analysis.

Single source shortest paths: Dijkstra's Algorithm.</td></tr>
<tr><td colspan="2">Laboratory Component:

Write & Execute C++/Java Program

1. To solve Knapsack problem using Greedy method.
2. To find shortest paths to other vertices from a given vertex in a weighted connected graph, using Dijkstra's algorithm.
3. To find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm. Use Union-Find algorithms in your program.
4. To find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.</td></tr>
<tr><td>Teaching-Learning Process</td><td>1. Chalk & board, Active Learning, MOOC, Problem based Learning.
2. Laboratory Demonstration.</td></tr>
<tr><td colspan="2">Module-4</td></tr>
<tr><td colspan="2">Dynamic Programming: All Pairs Shortest Paths: Floyd's Algorithm,Knapsack problem,

Travelling Sales Person problem.</td></tr>
</table>

Design and Analysis of Algorithms -21CS42- Laboratory Component

Laboratory Component: Write C++/ Java programs to 1. Solve All-Pairs Shortest Paths problem using Floyd's algorithm. 2. Solve Travelling Sales Person problem using Dynamic programming. 3. Solve 0/1 Knapsack problem using Dynamic Programming method.	
Teaching-Learning Process	1. Chalk & board, Active Learning, MOOC, Problem based Learning. 2. Laboratory Demonstration.
Module-5	
Backtracking: Sum of subsets problem, Hamiltonian cycles Problems.	
Laboratory Component:	
1. Design and implement C++/Java Program to find a subset of a given set S = {Sl, S2,…, Sn} of n positive integers whose SUM is equal to a given positive integer d. For example, if S = {1, 2, 5, 6,8} and d= 9, there are two solutions {1, 2, 6} and {1, 8}. Display a suitable message, if the given problem instance doesn't have a solution. 2. Design and implement C++/Java Program to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using backtracking principle.	
Teaching-Learning Process	1. Chalk & board, Active Learning, MOOC, Problem based learning. 2. Laboratory Demonstration.
Course outcome (Course Skill Set) At the end of the course the student will be able to: CO 1. Analyze the performance of the algorithms, state the efficiency using asymptotic notations andanalyze mathematically the complexity of the algorithm. CO 2. Apply divide and conquer approaches and decrease and conquer approaches in solving the problems analyze the same CO 3. Apply the appropriate algorithmic design technique like greedy method, transform and conquer approaches and compare the efficiency of algorithms to solve the given problem. CO 4. Apply and analyze dynamic programming approaches to solve some problems. and improve an algorithm time efficiency by sacrificing space. CO 5. Apply and analyze backtracking, branch and bound methods and to describe P, NP and NP-Complete problems.	

Design and Analysis of Algorithms -21CS42- Laboratory Component

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures not less than 35% (18 Marks out of 50) in the semester-end examination (SEE), and a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together

Continuous Internal Evaluation:

Three Unit Tests each of **20 Marks (duration 01 hour**)

1. First test at the end of 5th week of the semester
2. Second test at the end of the 10th week of the semester
3. Third test at the end of the 15th week of the semester

Two assignments each of **10 Marks**

4. First assignment at the end of 4th week of the semester
5. Second assignment at the end of 9th week of the semester

Practical Sessions need to be assessed by appropriate rubrics and viva-voce method. This will contribute to **20 marks**.

- Rubrics for each Experiment taken average for all Lab components – 15 Marks.

Design and Analysis of Algorithms -21CS42- Laboratory Component

- Viva-Voce– 5 Marks (more emphasized on demonstration topics)

The sum of three tests, two assignments, and practical sessions will be out of 100 marks and will be **scaled down to 50 marks**
(to have a less stressed CIE, the portion of the syllabus should not be common /repeated for any of the methods of the CIE. Each method of CIE should have a different syllabus portion of the course).

CIE methods /question paper has to be designed to attain the different levels of Bloom'staxonomy as per the outcome defined for the course.

Semester End Examination:

Theory SEE will be conducted by University as per the scheduled timetable, with common questionpapers for the subject (**duration 03 hours)**

1. The question paper will have ten questions. Each question is set for 20 marks. Marks scoredshall be proportionally reduced to 50 marks
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), **should have a mix of topics** under that module.

The students have to answer 5 full questions, selecting one full question from each module

Activity Based Learning (Suggested Activities in Class)/ Practical Based learning

1. Real world problem solving and puzzles using group discussion. E.g., Fake coin identification, Peasant, wolf, goat, cabbage puzzle, Konigsberg bridge puzzle etc.,
2. Demonstration of solution to a problem through programming.

The steps you need to follow to install Eclipse IDE in windows 10:

Eclipse for Java: How To Install Eclipse and Get Started with Java Programming on Windows

Eclipse (@ www.eclipse.org) is an *open-source* Integrated Development Environment (IDE) supported by IBM. Eclipse is popular for Java application development (Java SE and Java EE) and Android apps. It also supports C/C++, PHP, Python, Perl, and other web project developments via extensible plug-ins. Eclipse is cross-platform and runs under Windows, Linux and macOS.

Eclipse Versions

There are various versions of eclipse from Eclipse 1.0 to Eclipse 2019-12 (4.14).

1. **How to Install Eclipse IDE 2019-12 for Java Developers**

   How to install eclipse on windows 10 64 bit for java

   https://www.youtube.com/watch?v=rSFraftR4I4

   Eclipse with java programming installation steps

   https://www3.ntu.edu.sg/home/ehchua/programming/howto/EclipseJava_HowTo.html

   Video

   https://www.youtube.com/watch?v=WIzzHeWukUU

Step 0: Install JDK

To use Eclipse for Java programming, you need to first install Java Development Kit (JDK). Read "How to Install JDK for Windows".

1: Open Chrome Browser

2: In google.com search **java jdk**

3: Click **Java SE - Downloads | Oracle Technology Network | Oracle**

4: In Java SE Downloads**,** click **JDK download** under Oracle JDK

5: select **Windows x64 Compressed Archive** in the list i.e click **jdk-13.0.2_windows-x64_bin.zip** [178.99 MB]

6**:** Unzip in the drive you want to install preferably in C:\ drive

Step 1: Download

Download Eclipse from https://www.eclipse.org/downloads. Under "Get Eclipse IDE 2019-12" ⇒ Click "Download s". For beginners, choose the "**Eclipse IDE for Java Developers**" and "**Windows 64-bit**" (e.g., "eclipse-java-2019-12-R-win32-x86_64.zip" - about 201MB) ⇒ Download.

Step 2: Unzip

To install Eclipse, simply unzip the downloaded file into a directory of your choice (e.g., "c:\myProject").

The zip version is preferred because there is no need to run any installer. Moreover, you can simply delete the entire Eclipse directory when it is no longer needed (without running any un-installer). You are free to move or rename the directory. You can install (unzip) multiple copies of Eclipse in the same machine.

2. Writing your First Java Program in Eclipse

Step 0: Launch Eclipse

1. Launch Eclipse by running "eclipse.exe" from the Eclipse installed directory.
2. Choose an appropriate directory for your *workspace*, i.e., where you would like to save your files (e.g., c:\myProject\eclipse for Windows) ⇒ Launch.
3. If the "Welcome" screen shows up, close it by clicking the "close" button next to the "Welcome" title.

Step 1: Create a new Java Project

For each Java application, you need to create a *project* to keep all the source files, classes and relevant resources.

To create a new Java project:

1. Choose "File" menu ⇒ "New" ⇒ "Java project" (or "File" ⇒ "New" ⇒ "Project" ⇒ "Java project").
2. The "New Java Project" dialog pops up.

a. In "Project name", enter "FirstProject".

b. Check "Use default location".

c. In "JRE", select "Use default JRE (currently 'JDK11.0.x')". But make sure that your JDK is 1.8 and above.

d. In "Project Layout", check "Use project folder as root for sources and class files".

Push "Finish" button.

In "Create module-info.java" dialog, Click "Don't Create". (Note: For easier version of Eclipse, you need to use the "Next" Button in the previous step, and uncheck "Create module-info.java file").

Step 2: Write a Hello-world Java Program

1. In the " Explorer" (left pane) ⇒ Right-click on "FirstProject" (or use the "File" menu) ⇒ New ⇒ Class.
2. The "New Java Class" dialog pops up.
   a. In "Source folder", keep the "FirstProject".
   b. In "", delete the content if it is not empty.
   c. In "Name", enter "Hello".
   d. Check "public static void main(String[] args)".
   e. Don't change the rest.

Push "Finish" button.

The source file "Hello.java" opens on the editor panel (the center pane). Enter the following codes:

```
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }
}
```

Step 3: Compile & Execute the Java Program

1. There is no need to *compile* the Java source file in Eclipse explicitly. It is because Eclipse performs the so-called *incremental compilation*, i.e., the Java statement is compiled as and when it is entered.
2. To run the program, right-click anywhere on the source file "Hello.java" (or choose "Run" menu) ⇒ Run As ⇒ Java Application.
3. The output "Hello, world!" appears on the Console panel (the bottom pane).

NOTES:

- You should create a NEW Java project for EACH of your Java application.
- Nonetheless, Eclipse allows you to keep more than one programs in a project, which is handy for writing toy programs (such as your tutorial exercises). To run a particular program, open and right-click on the source file ⇒ Run As ⇒ Java Application.
- Clicking the "Run" button (with a "Play" icon) runs the recently-run program (based on the previous configuration). Try clicking on the "down-arrow" besides the "Run" button.

2.1 Correcting Syntax Errors

Eclipse performs incremented compilation, as and when a source "line" is entered. It marked a source line having syntax error with a RED CROSS. Place your cursor at the RED CROSS to view the error message.

You CANNOT RUN the program if there is any syntax error (marked by a RED CROSS before the filename). Correct all the syntax errors; and RUN the program.

HINTS: In some cases, Eclipse shows a ORANGE LIGHT-BULB (for HINTS) next to the ERROR RED-CROSS (Line 5 in the above diagram). You can click on the LIGHT-BULB to get a list of HINTS to resolve this particular error, which may or may not work!

SYNTAX WARNING: marked by a orange triangular exclaimation sign. Unlike errors, warnings may or may not cause problems. Try to fix these warnings as well. But you can RUN your program with warnings.

3. Read the Eclipse Documentation

At a minimum, you SHOULD browse through Eclipse's "**Workbench User Guide**" and "**Java Development User Guide**" - accessible via the Eclipse's "Welcome" page or "Help" menu. This will save you many agonizing hours trying to figure out how to do somethings later.

4. Debugging Programs in Eclipse

Able to use a graphics debugger to debug program is crucial in programming. It could save you countless hours guessing on what went wrong.

Step 0: Write a Java Program

The following program computes and prints the factorial of *n* (=1*2*3*...**n*). The program, however, has a logical error and produce a wrong answer for *n*=20 ("The Factorial of 20 is -2102132736" - a negative number?!).

```
1   /** Compute the Factorial of n, where n=20.
2    *  n! = 1*2*3*...*n
3    */
4   public class Factorial {
5     public static void main(String[] args) {
6       int n = 20;         // To compute factorial of n
7       int factorial = 1;  // Init the product to 1
8
9       int i = 1;
10      while (i <= n) {
11        factorial = factorial * i;
12        i++;
13      }
14      System.out.println("The Factorial of " + n + " is " + factorial);
15    }
16  }
```

Let's use the graphic debugger to debug the program.

Step 1: Set an Initial Breakpoint

A *breakpoint* suspends program execution for you to examine the internal states (e.g., value of variables) of the program. Before starting the debugger, you need to set at least one breakpoint to suspend the execution inside the program. Set a breakpoint at main() method by double-clicking on the *left-margin* of the line containing main(). A *blue circle* appears in the left-margin indicating a breakpoint is set at that line.

Step 2: Start Debugger

Right click anywhere on the source code (or from the "Run" menu) ⇒ "Debug As" ⇒ "Java Application" ⇒ choose "Yes" to switch into "Debug" perspective (A *perspective* is a particular arrangement of panels to suits a certain development task such as editing or debugging). The program begins execution but suspends its operation at the breakpoint, i.e., the main() method.

As illustrated in the following diagram, the highlighted line (also pointed to by a blue arrow) indicates the statement to be executed in the *next* step.

Step 3: Step-Over and Watch the Variables and Outputs

Click the "Step Over" button (or select "Step Over" from "Run" menu) to *single-step* thru your program. At each of the step, examine the value of the variables (in the "Variable" panel) and the outputs produced by your program (in the "Console" Panel), if any. You can also place your cursor at any variable to inspect the content of the variable.

Single-stepping thru the program and watching the values of internal variables and the outputs produced is the *ultimate* mean in debugging programs - because it is exactly how the computer runs your program!

Step 4: Breakpoint, Run-To-Line, Resume and Terminate

As mentioned, a breakpoint *suspends* program execution and let you examine the internal states of the program. To set a breakpoint on a particular statement, double-click the left-margin of that line (or select "Toggle Breakpoint" from "Run" menu).

"Resume" continues the program execution, up to the next breakpoint, or till the end of the program.

"Single-step" thru a loop with a large count is time-consuming. You could set a breakpoint at the statement immediately outside the loop (e.g., Line 11 of the above program), and issue "Resume" to complete the loop.

Alternatively, you can place the cursor on a particular statement, and issue "Run-To-Line" from the "Run" menu to continue execution up to the line.

"Terminate" ends the debugging session. Always terminate your current debugging session using "Terminate" or "Resume" till the end of the program.

Step 5: Switching Back to Java perspective

Click the "Java" perspective icon on the upper-right corner to switch back to the "Java" perspective for further programming (or "Window" menu ⇒ Open Perspective ⇒ Java).

Important: Mastering the use of debugger is crucial in programming. Explore the features provided by the debuggers.

Other Debugger's Features

Step-Into and Step-Return: To debug a *method*, you need to use "Step-Into" to step into the *first* statement of the method. ("Step-Over" runs the function in a single step without stepping through the statements within the function.) You could use "Step-Return" to return back to the caller, anywhere within the method. Alternatively, you could set a breakpoint inside a method.

Modify the Value of a Variable: You can modify the value of a variable by entering a new value in the "Variable" panel. This is handy for temporarily modifying the behavior of a program, without changing the source code.

5. Tips & Tricks

5.1 General Usages (for all Programming Tasks)

These are the features that I find to be most useful in Eclipse:

1. **Maximizing Window (Double-Clicking):** You can double-click on the "header" of any panel to *maximize* that particular panel, and double-click again to *restore* it back. This feature is particularly useful for writing source code in full panel.
2. **Shorthand Templates (sysout, for,...):** You can type "sysout" followed by a ctrl+space (or alt-/) as a shorthand for typing "System.out.println()". The default shortcut key (ctrl-space or alt-/) depends on the system. Check your system's

shortcut key setting in "Edit" ⇒ "Content Assist" ⇒ "Default". Take note that many of you use ctrl+space to switch between input languages. You need to reconfigure either your language switching hot-key or Eclipse.
Similarly, you can type "for" followed by ctrl-space (or alt-/) to get a for-loop. You can create your own shorthand in "Window" menu ⇒ "Preferences" ⇒ "Java" ⇒ "Editor" ⇒ "Templates". (Alternatively, in "Window" ⇒ "Preferences" ⇒ type "template" as filter text and choose "Java" ⇒ "Editor" ⇒ "Templates".)
You can change your key settings in "Window" menu ⇒ "Preferences" ⇒ "General" ⇒ "Key" ⇒ choose "Command", "Content Assist". (Alternatively, in "Window" ⇒ "Preferences" ⇒ type "key" as filter text and choose "General" ⇒ "Key".)

3. **Intelli-Sense (ctrl-space):** You can use ctrl-space to activate the "intelli-sense" (or content assist). That is, Eclipse will offer you the choices, while you are typing.
4. **Source Formatting (ctrl-shift-f):** Right-click on the source. Choose "Source" ⇒ "Format" to let Eclipse to layout your source codes with the proper indentation.
5. **Source Toggle Comment (ctrl-/):** To comment/uncomment a block of codes, choose "Source" ⇒ "Toggle Comment".
6. **Hints for Correcting Syntax Error:** If there is a syntax error on a statement, a red mark will show up on the left-margin on that statement. You could click on the "light bulb" to display the error message, and also select from the available hints for correcting that syntax error.
7. **Refactor (or Rename) (alt-shift-r):** You can rename a variable, method, class, or even the project easily in Eclipse. Select and right-click on the entity to be renamed ⇒ "Refactor" ⇒ "Rename". Eclipse can rename all the occurrences of the entity.
8. **Line Numbers:** To show the line numbers, choose "Window" menu ⇒ "Preferences" ⇒ "General" ⇒ "Editors" ⇒ "Text Editors" ⇒ Check the "Show Line Numbers" Box. You can also configure many editor options, such as the number of spaces for tab. Alternatively, you can right-click on the left-margin, and check "Show Line Numbers".
9. **Error Message Hyperlink:** Click on an error message will hyperlink to the corresponding source statement.
10. **Changing Font Type and Size:** From "Window" menu ⇒ "Preferences" ⇒ "General" ⇒ "Appearance" ⇒ "Colors and Fonts" ⇒ expand "Java" ⇒ "Java Editor Text Font" ⇒

"Edit". (Alternatively, in "Window" ⇒ "Preferences" ⇒ type "font" as filter text and choose the appropriate entry.)

11. **Unicode Support:** To enable Unicode support, select "Window" menu ⇒ Preferences ⇒ General ⇒ Workspace ⇒ Text file encoding ⇒ UTF-8. This sets the default character set used for file encoding, similar to VM's command-line option -Dfile.encoding=UTF-8. Commonly used charsets for Unicode are UTF-8, UTF-16 (with BOM), UTF-16BE, UTF-16LE. Other charsets are US-ASCII, ISO-8859-1.
12. **Mouse Hover-over:** In debug mode, you could configure to show the variable's value when the mouse hovers over the variable. Select "Window" menu ⇒ "Preferences" ⇒ "Java" ⇒ "Editor" ⇒ "Hover".
13. **Comparing Two Files:** In " Explorer", select two files (hold the control key) ⇒ Right-click ⇒ Compare with.
14. **Useful Eclipse Shortcut Keys:**
    - F3: Goto the declaration of the highlighted variable/method.
    - Ctrl-Shift-G: Search for ALL references of the highlighted variable/method in workspace.
    - Ctrl-G: Search for the Declaration of a variable/method in workspace. Don't use Find (Ctrl-F), but use the above context-sensitive search.
    - Ctrl-Shift-F: Format the source code.
    - Ctrl-Shift-O: Organize imports.
    - Alt-Shift-R: Rename. (Don't use Find/Replace.)
    - Ctrl-Space: auto-complete.
15. **Explorer vs. Navigator:** We usually use " Explorer" in programming, but it will not show you all the folders and files under the project. On the other hand, "Navigator" is a file manager that shows the exact file structure of the project (similar to Windows Explorer). You can enable the Navigator by "Window" ⇒ Show view ⇒ Navigator.
16. **Spell Check:** To enable spell check, select Window ⇒ Preferences ⇒ type "spell" in the filter ⇒ General ⇒ Editors ⇒ Text Editors ⇒ Spelling ⇒ Check "Enable spell checking".

Also provide a "User defined dictionary" (with an initially empty text file). To correct mis-spell words, right-click and press ctrl-1 (or Edit menu ⇒ Quick Fix).

17. **Eclipse's Log File:** Goto Help ⇒ about Eclipse ⇒ Installation details ⇒ Configuration ⇒ View Error Log.
18. **Viewing two files in split screen:** Simply click and hold on the title of one file and drag it to the lower side of the screen. [To view the same file on split screen, create a new editor window by selecting Window ⇒ New Editor; and drag one window to the lower side of the screen.]
19. **Block Select (Column Select):** Push Alt-Shift-A to toggle between block-select mode and normal mode.
20. **Snippets:**
    - To view the snippet window: choose "Window" ⇒ Show View ⇒ Snippets.
    - To create a new snippet category: Right-click ⇒ Customize ⇒ New.
    - To create a new snippet item: Copy the desired text ⇒ Select the snippet category ⇒ paste as snippet.
    - To insert a snippet: place the cursor on the desired location at the editor panel ⇒ click the snippet item.
21. **Word Wrap (Line Wrap):** Word-wrap (or line-wrap) is essential for editing long HTML documents without the horizontal scroll bar. However, the Eclipse's HTML Editor and Text Editor do not support word-wrap.

    You could install a plug-in called "Word Wrap" from http://ahtik.com/eclipse-update/. Choose "Help" ⇒ Install New Software ⇒ in "Work with" Enter "http://ahtik.com/eclipse-update/".

    To activate word wrap, right-click on the editor panel ⇒ select "Word Wrap".
22. **Creating "link folder" in project**: You do not have to place all the folders under the project base directory, instead, you can use so-called "link folders" to link to folder outside the project base directory.

To create a link folder in a project, right-click on the project ⇒ File ⇒ New ⇒ Folder ⇒ Advanced ⇒ Check Link to alternate Location (Linked Folder).

23. **Running Eclipse in "clean" mode:** You can run eclipse in so-called "clean" mode, which wipes all the cached data and re-initialize the cache, by running eclipse from command-line with "-clean" argument (i.e., "eclipse -clean"). It is useful if something is not working proper, especially if you install a new copy of Eclipse.
24. Show the Right Margin: Window ⇒ Preferences ⇒ General ⇒ Editors ⇒ Text Editors ⇒ Show Print Margin and set the column number.
25. Let me know if you have more tips to be included here.

5.2 Update Eclipse and Install new Software

1. **Install New Software:** Select "Help" menu ⇒ Install New Software ⇒ In "Work With", pull down the select menu and choose a software site.
2. **Update:** Select "Help" menu ⇒ Check for Updates ⇒.

5.3 For Java Application Development Only

1. **Small Toy Java Programs:** You can keep many small programs (with main()) in one Java project instead of create a new project for each toy program. To run the desired program, right-click on the source file ⇒ "Run as" ⇒ "Java Application".
2. **Scanner/printf() and JDK 1.5:** If you encounter syntax error in using printf() or Scanner (which are available from JDK 1.5), you need to check your compiler settings. Select "Window" menu ⇒ Preferences ⇒ open the "Java" node ⇒ select "Compiler" ⇒ in "Compiler compliance level" ⇒ select the latest release, which should be "1.5" or above.
3. **Command-Line Arguments:** To provide command-line arguments to your Java program in Eclipse, right-click on the source file ⇒ "Run Configurations" ⇒ Under the "Main" panel, check that "Project" name and "Main Class" are appropriate ⇒ Select the "Argument" tab ⇒ type your command-line arguments inside the "Program Arguments" box ⇒ "Run".
4. **Resolving Import (Ctrl-Shift-o)**: To ask Eclipse to insert the import statements for classes. Useful when you copy a large chunk of codes without the corresponding import statements.

5. **Including Another Project:** To include another project in the same work space, right-click on the project ⇒ Build Path ⇒ Configure Build Path... ⇒ Select "Projects" tab ⇒ "Add..." to select project in the existing work space ⇒ OK.
6. **Exporting a Project to a JAR file:** Right-click on the project ⇒ Export... ⇒ Java, JAR File ⇒ Next ⇒ Select the files to be exported ⇒ Next ⇒ Next ⇒ In "JAR Manifest Specification" dialog, enter the main class (if you wish to run the JAR file directly) ⇒ Finish.
7. **Unit Testing:** If you keep your test in another project, you need to include the project under test in your Build Path (see above).

To create a test case: Right-click on the project ⇒ New ⇒ JUnit Test Case ⇒ the "New JUnit Test Case" dialog appears. Select "New JUnit 4 Test". In "Name", enter your class name. In "Class under test", browse and select the class to be tested.
To run the test: Right-click ⇒ "Run As" ⇒ "JUnit Test". The results are displayed in a special "JUnit console".

8. **Adding External JAR files & Native Libraries (".dll", ".lib", ".a", ".so"):** Many external Java s (such as JOGL, Java3D, JAMA, etc) are available to extend the functions of JDK. These s typically provide a "lib" directory containing JAR files (".jar") (Java Archive - a single-file of Java classes) and native libraries (".dll", ".lib" for windows, ".a", ".so" for Linux and macOS).
   To include these external s into an Eclipse's project, right-click on the project ⇒ Build Path ⇒ Add External Archives ⇒ Navigate to select the JAR files (".jar") to be included. In " Explorer", right-click on the JAR file added ⇒ Properties:
   - To include native libraries (".dll", ".lib", ".a", ".so"), select "Native Library" ⇒ "Location Path" ⇒ "External Folder".
   - To include the javadoc, select "JavaDoc Location" ⇒ "JavaDoc URL" ⇒ You can specify a *local* file or a remote link.
   - To include source file (for debugging), select "Java Source Attachment".

   All the above options are also accessible via project's property ⇒ "Build Path".
   Notes: The JAR files must be included in the CLASSPATH. The native library directories must be included in JRE's property "java.library.path", which normally but not necessarily includes all the paths from the PATH environment variable. Read "External JAR files and Native Libraries".
9. **Creating a User Library:** You can also create a Eclipse's *user library* to include a set of JAR files and native libraries, that can then be added into subsequent Eclipse projects. For example, I created a user library for "JOGL" as follows:
   a. From "Window" menu ⇒ Preferences ⇒ Java ⇒ Build Path ⇒ User Libraries ⇒ New ⇒ In "User library name", enter "jogl". The "User Library" dialog appears.
   b. In "User Library" dialog ⇒ Select "jogl" ⇒ Add JAR... ⇒ Navigate to <JOGL_HOME>/lib, and select "gluegen-rt.jar" and "jogl.jar".

c. Expand the "jogl.jar" node ⇒ Select "Native library location: (none)" ⇒ Edit... ⇒ External Folder... ⇒ select <JOGL_HOME>/lib.

d. Expand the "jogl.jar" node ⇒ Select "Javadoc location: (none)" ⇒ Edit... ⇒ Javadoc in archive ⇒ In "Archive Path", "Browse" and select the downloaded JOGL API documentation zip-file ⇒ In "Path within archive", "Browse" and expand the zip-file to select the top-level path (if any) ⇒ Validate. Alternatively, you can provide the path to the un-zipped javadocs. This is needed for Eclipse to display javadoc information about classes, fields, and methods.

e. You may provide the source files by editing "Source attachment: (none)". Source is needed only if you are interested to debug into the JOGL source codes.

For EACH subsequent Java project created that uses JOGL, right-click on the project ⇒ Build Path ⇒ Add Libraries ⇒ Select "User Library" ⇒ Check "jogl".

Running an External Program: Suppose that you want to run a Perl script on the selected file, you can configure an external tool as follows:

. From "Run" menu ⇒ External Tools ⇒ External Tools Configuration... ⇒ The "External Tools Configuration" dialog appears.

a. In "Name", enter your tool name.

b. Choose the "Main" tab ⇒ In "Location", "Browse File System..." to choose the perl interpreter "perl" ⇒ In "Arguments", enter "path/scriptname.pl ${resource_loc}", where ${resource_loc} is an Eclipse variable that denotes the currently selected resource with absolute path.

c. Choose the "Common" tab ⇒ In "Standard Input and Output", uncheck "Allocate Console", check "File" and provide an output file (e.g., d:\temp\${resource_name}.txt).

d. (If you use the CYGWIN perl interpreter, need to set environment variable CYGWIN=nodosfilewarning to disable warning message.)

To run the configured external tool, select a file ⇒ run ⇒ external tool ⇒ tool name.

Viewing Hex Code of Primitive Variables in Debug mode: In debug perspective, "Variable" panel ⇒ Select the "menu" (inverted triangle) ⇒ Java ⇒ Java

Preferences... ⇒ Primitive Display Options ⇒ Check "Display hexadecimal values (byte, short, char, int, long)".

Adding a New Version of JDK/JRE: First, you can check the installed JDK/JRE via "Window" menu ⇒ "Preferences" ⇒ Expand "Java" node ⇒ "Installed JREs". Check the "Location" current JRE installed to make sure that it is the intended one. You can use the "Add" button to add a new version of JRE. For program development, I recommend that you add the JDK (instead of JRE). [The "Location" decides the extension directory used for including additional JAR files, e.g., $JAVA_HOME\jre\lib\ext.]

Design and Analysis of Algorithms -21CS42- Laboratory Component

Module-1

1. **Sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the brute force method works along with its time complexity analysis: worst case, average case and best case.**

Selection Sort Using Brute Force Method:

Introduction

Sorting data is a frequent problem in computer science. Given a collection of elements, the goal is to rearrange them in some order. Common examples are sorting an array alphabetically or from smallest to largest.

Sorted data is a lot easier to manipulate. Finding the largest or smallest element of an array can be done in constant time if the array is sorted. Searching for an element is a lot faster using algorithms such as Binary Search which rely on the assumption that the array is already sorted.

One of the simplest algorithms for sorting data is **Selection Sort**.

Selection Sort

Selection sort is an in-place comparison sorting algorithm that uses brute force to sort an array.

In-place means that the algorithm uses a small constant amount of space for extra storage.

It's called a "brute force" algorithm because it uses the simplest and most ineffective way of calculating the solution. However, it does makes up for it with its straightforward implementation.

The algorithm divides the array into two subarrays:

- A sorted subarray
- An unsorted subarray

The sorted subarray is empty in the beginning. In every iteration, the smallest element of the unsorted array will be appended to the end of the sorted array by swapping. This way, the sorted array will eventually contain all the elements of the original array.

An example array we want to sort in ascending order:

Sorted array	Unsorted array	Minimal element of the unsorted array
[]	[16, 5, 30, 6, 2, 7]	2
[2]	[16, 5, 20, 6, 7]	5
[2, 5]	[16, 20, 6, 7]	6
[2, 5, 6]	[16, 7, 20]	7
[2, 5, 6, 7]	[16, 20]	16
[2, 5, 6, 7, 16]	[20]	20
[2, 5, 6, 7, 16, 20]	[]	

Implementation

The selectionSort() method takes just one argument, the array that needs to be sorted. We'll iterate through the unsorted array, which will be between indexes i and j, find it's minimum and place it into the sorted array by swapping:

Pseudocode :

```
selectionSort(int[] nums)
 {
   for (int i ← 0; i < nums.length; i++)
  {
      // min is the index of the smallest element with an index greater or equal to i
      int min ← i;
      for (int j← i + 1; j < nums.length; j++)
            {
                  if (nums[j] < nums[min])
                  {
                   min ← j;
                  }
            }
      // Swapping i-th and min-th elements
      swap nums[i] with nums[min];
    }
}
```

Selection Sort Time Complexity

Time complexity is a way to describe how much time an algorithm needs to finish executing relative to the size of the input. Analyzing the time it takes for an algorithm to give output is of crucial importance. Imagine a telephone book application that would take a day to sort all the numbers after a new number was added. That would be way less useful than the same app that would do it almost instantly.

Performance depends on the hardware as well as software, but the same program can be run on many different types of hardware. The Big-O notation makes it easier to approximate the time needed for a program to execute, regardless of software.

The average and worst-case time complexity of Selection Sort is $O(n^2)$. This makes Selection Sort a lot slower than many other comparison sorting algorithms like Merge Sort or Insertion Sort which have the worst-case time complexity *(O(nlogn))*. Interestingly, *O(nlogn)* is the best that can be achieved by any comparison sorting algorithm.

Time Complexity Analysis

Showing that Selection Sort has quadratic time complexity comes down to calculating the number of times the inner loop will be iterated. We can see this if we go through the code line by line and try to approximate the time it takes to execute each line of code:

```
for(int i=0; i<nums.length; i++){
```

Everything in the inner block of the loop will be executed n times, where n is the length of a given array:

```
int min= i;
```

min will be initialized to i exactly n times. Now comes the tricky part:

```
    for (int j = i + 1; j < nums.length; j++) {
```

Since this loop is nested, it takes a bit of math to calculate the number of times the block of code inside it will execute. Let's work it out.

When i is equal to 0, j will go from 1 to n, meaning every instruction in the inner block will execute n times. When i increases to 1, j will stay between 2 and n, which implies the inner block will execute n-2 times. Summing this up:

(n - 1) + (n - 2) + ... + 1

The sum of a sequence of natural numbers is calculated using something called Gauss's trick, and it results in $(n^2 - n)/2$. Simplifying this, results in $O(n^2)$ time complexity.

Put simply, when calculating the complexity of an algorithm O(f(n)), we need to look for the highest power of n in the function f(n) and isolate it. This is because any part of the equation that has a lower power will not affect the result in any significant way.
For example, we have the function $f(x) = x^2+13x+23$
O(f(x)) would be the highest power of x in the equation, which in this case is x^2.

Here's how it performed after sorting an array containing 10,000 integers in random order:

```
public static void main(String[] args) {
    int[] array = new int[10000];
    for (int i = 0; i < array.length; i++) {
        array[i] = i;
    }

    // Shuffle array
    Collections.shuffle(Arrays.asList(array));

    // Print shuffled collection
    System.out.println(Arrays.toString(array));

    long startTime = System.nanoTime();
    selectionSort(array);
    long endTime = System.nanoTime();

    // Print sorted collection
    System.out.println(Arrays.toString(array));

    // Print runtime in seconds
    System.out.println("Selection Sort runtime: " + (endTime - startTime)/1000000000);
}
```

Running it 10 times, this code produced the following results:

Time(s)	Selection Sort
First Run	0.024
Second Run	0.020
Third Run	0.022
Fourth Run	0.020
Fifth Run	0.025
Sixth Run	0.022
Seventh Run	0.021
Eight Run	0.031
Ninth Run	0.022
Tenth Run	0.029

The average run time was *0.0236* seconds, though; this will majorly depend on your machine as well.

Selection Sort Space Complexity

Space Complexity is also a big factor in algorithm design. Our programs are bound, not only by the time that they need to execute but also by memory usage. There is a limited amount of memory on any computer, so a programmer should keep an eye on that too.

The space complexity of Selection Sort is constant(*O(1)*) because it is in-place, which is great. Worst case complexity of Selection Sort is, unfortunately, $O(n^2)$ as well, which means that even if the algorithm gets an already sorted array as input, it will still take a lot of time to return the unchanged array.

This algorithm has decent performance if the collection doesn't have a lot of elements. If the array has ~10 elements, the difference in performance between different sorting algorithms shouldn't be that noticeable, and Selection Sort might even outperform other divide-and-conquer algorithms.

Where Selection Sort shines, is when the number of swaps needs to be minimal. In the worst case, there will only be n-1 swaps, which is the minimal possible number of swaps that need to be performed. This is quite intuitive if consider that every element will be placed in its right spot in the sorted array right away.

Conclusion

Selection Sort is a brute force in-place comparison sort which continuously finds the minimum of an unsorted subarray and places it in the correct position in the sorted subarray. Due to its simplicity, it's often one of the first algorithms that are taught in computer science courses all around the world.

Even if more efficient algorithms come built-it, it's still important to understand the underlying logic and complexity analysis to avoid common issues and to make sure that the tool being used is the one that's best suited for the job at hand.

Design and Analysis of Algorithms -21CS42- Laboratory Component

Program1:

```
import java.util.Random;

import java.util.Scanner;

public class Selsort
{
        public static void selectionSort(int[] nums)
        {
                for (int i = 0; i < nums.length; i++)
                {
                 // min is the index of the smallest element with an index greater or equal to i
                        int min = i;
                        for (int j = i + 1; j < nums.length; j++)
                        {
                           if (nums[j] < nums[min])
                           {
                             min = j;
                            }
                        }
                 // Swapping i-th and min-th elements
                int swap = nums[i];
                nums[i] = nums[min];
                nums[min] = swap;
                }
        }


public static void main(String[] args)

{

   int[] array = new int[10000];

   for (int i = 0; i < array.length; i++) {

        array[i] = i;

   }

   // Shuffle array

   Collections.shuffle(Arrays.asList(array));


   // Print shuffled collection

   System.out.println(" Unsorted Array List is:")

   System.out.println(\tArrays.toString(array));
```

```
            long startTime = System.nanoTime();
            System.out.println(“Selection Sort”);
             selectionSort(array);
              long endTime = System.nanoTime();

             // Print sorted collection
            System.out.println(“ the Sorted Array is:”);
            System.out.println(\tArrays.toString(array));

             // Print runtime in seconds
   System.out.println("Selection Sort runtime: " + (endTime - startTime)/1000000000);
}
```

Output:

Unsorted Array List is: ---------

the Sorted Array is: 0 1 2 3…..10000

Selection Sort runtime: ---

a) **The elements generated using the random number generator**

```
public static void main(String args[])
{
        static int max=50000;
        int n,i;
        Scanner in=new Scanner(System.in);
        Random rand=new Random();
        System.out.println("Selectionsort Test");
        /* Accept no.of Elements */
        System.out.println("\nEnter the number of elements");
        n=in.nextInt();
        /* create array of n elements */
```

```
        int arr[]=new int[max];
        try
                {
                        /* Generate Random Numbers */
                        for(i=0;i<n;i++)
                        arr[i]=rand.nextInt(100);
                        /* Print random numbers */
                        System.out.println("\nthe random elements are ");
                        for(i=0;i<n;i++)
                        System.out.println(arr[i]+" ");
                        long start_time=System.nanoTime();
                        /*call method Selection Sort*/
                                selectionSort(arr);
                        long end_time=System.nanoTime();
                        /* Print Sorted Array */
                        System.out.println("\nThe Elements After sorting");
                        for(i=0;i<n;i++)
                        System.out.println(arr[i]+" ");
                        long t=end_time - start_time;
                        System.out.println("Time taken for execution is:"+t+" nanoseconds);
                }
        catch(ArrayIndexOutOfBoundsException ae)
                {
                        System.out.println("Array Index reached maximum ");
                }
}
```

Output:

```
Selectionsort Test
Enter the number of elements
10
the random elements are
17
31
```

81
44
91
24
87
42
57
17
The Elements after sorting
17
17
24
31
42
44
57
81
87
91
Time taken for execution is: _____ nanoseconds

Note: For n=5000 to 50000 in step of 5000 note down the time in nanoseconds. Convert time in seconds and plot the graph with n as x axis and time as y axis.

b) The elements read from a file

Create a text file in notepad and enter random numbers one in each line. Save the file as t.txt

```
public static void main(String args[])
{
        int n,i;
        FILE *f1;
        int line[50];
        int limit;
        System.out.println("Selectionsort Test");
        /* Accept no.of Elements */
        System.out.println("\nEnter the number of elements");
        n=in.nextInt();
        /* create array of n elements */
        int arr[]=new int[max];
```

```
        try{
        /* Elements from file are: */
        f1= fopen("t.txt","r");
while(i<= n-1) && (fgets(line,50,f1)!=NULL))
{
fscanf(line,"%d",&limit);
arr[i] = limit;
i++;
}
fclose(f1);
        /* Print random numbers */
        System.out.println("\nthe random elements are ");
        for(i=0;i<n;i++)
        System.out.println(arr[i]+" ");
        long start_time=System.nanoTime();
        /*call method Selection Sort*/
          selectionSort(arr);
        long end_time=System.nanoTime();
        /* Print Sorted Array */
        System.out.println("\nThe Elements After sorting");
        for(i=0;i<n;i++)
        System.out.println(arr[i]+" ");
        long t=end_time - start_time;
        System.out.println("Time taken for execution is:"+t+" nanoseconds);
        }
        catch(ArrayIndexOutOfBoundsException ae)
        {
                System.out.println("Array Index reached maximum ");
        }
        }
}
```

Design and Analysis of Algorithms -21CS42- Laboratory Component

Module2

1. Sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the divide-and-conquer method works along with its time complexity analysis: worst case, average case and best case.

Quick Sort using Divide and Conquer Technique:

Quick Sort divides the array according to the value of elements. It rearranges elements of a given array A[0..n-1] to achieve its partition, where the elements before position s are smaller than or equal to A[s] and all the elements after position s are greater than or equal to A[s].

A[0]……..A[s-1] A[s] A[s+1]…A[n-1]

All are <= A[s] All are >= A[s]

Pseudocode : QUICKSORT(a[l..r])

```
//Sorts a subarray by quicksort
//Input: A subarray A[l..r] of A[0..n-1],defined by its left and right indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
{
        if l<r
        {
                s← Partition(A[l..r]) //s is a split position
                QUICKSORT(A[l..s-1])
                QUICKSORT(A[s+1..r])
        }
}
```

Pseudocode : Partition(A[l..r])

```
//Partition a subarray by using its first element as its pivot
//Input:A subarray A[l..r] of A[0..n-1],defined by its left and right indices l and r (l<r)
//Output:A partition of A[l..r],with the split position returned as this function's value
{
p ← A[l]
i ← l; j ← r+1
        repeat
        {
                repeat i ← i+1 until A[i] >=p  and i<=high
                repeat j ← j-1 until A[j] <=p
                swap(A[i],A[j])

        } until i>=j
```

```
        swap(A[i],A[j]) // undo last swap when i>=j
        swap(A[l],A[j])
        return j

}
```

Program 1

a)The elements generated using the random number generator

```
import java.util.Random;
import java.util.Scanner;


public class QuickSort
{
        static int max=50000;
        public static int partition(int a[],int low,int high)
        {
        int i,j,temp,key;
        key=a[low];
        i=low;
        j=high+1;
        while(i<=j)
        {
                do
                        i++;
                while (key>=a[i]&& i<=high);
                do
                        j--;
                while(key<a[j]);
                if(i<j)
                {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
```

```
                }
        }
        temp=a[low];
        a[low]=a[j];
        a[j]=temp;
        return j;
}
public static void qs (int a[],int low, int high)
{
        int mid;
        if(low<high)
        {
                mid=partition(a,low,high);
                qs(a,low,mid-1);
                qs(a,mid+1,high);
        }
}
public static void main(String args[])
{
        int n,i;
        Scanner in=new Scanner(System.in);
        Random rand=new Random();
        System.out.println("Quicksort Test");
        /* Accept no.of Elements */
        System.out.println("\nEnter the number of elements");
        n=in.nextInt();
        /* create array of n elements */
        int arr[]=new int[max];
        try{
        /* Generate Random Numbers */
        for(i=0;i<n;i++)
```

```
        arr[i]=rand.nextInt(100);
        /* Print random numbers */
        System.out.println("\nthe random elements are ");
        for(i=0;i<n;i++)
        System.out.println(arr[i]+" ");
        long start_time=System.nanoTime();
        /*call method Quick Sort*/
        qs(arr,0,n-1);
        long end_time=System.nanoTime();
        /* Print Sorted Array */
        System.out.println("\nThe Elements After sorting");
        for(i=0;i<n;i++)
        System.out.println(arr[i]+" ");
        long t=end_time - start_time;
        System.out.println("Time taken for execution is:"+t+" nanoseconds);
        }
        catch(ArrayIndexOutOfBoundsException ae)
        {
                System.out.println("Array Index reached maximum ");
        }
        }
}
```

Output:

```
Quicksort Test
Enter the number of elements
10
the random elements are
17
31
81
44
91
24
```

87
42
57
17
The Elements after sorting
17
17
24
31
42
44
57
81
87
91
Time taken for execution is:6316 nanoseconds

Note: For n=5000 to 50000 in step of 5000 note down the time in nanoseconds. Convert time in seconds and plot the graph with n as x axis and time as y axis.

b) The elements read from a file

Create a text file in notepad and enter random numbers one in each line. Save the file as t.txt

```
import java.util.Random;
import java.util.Scanner;

public class QuickSort
{
	static int max=50000;
	public static int partition(int a[],int low,int high)
	{
	int i,j,temp,key;
	key=a[low];
	i=low;
	j=high+1;
	while(i<=j)
	{
		do
			i++;
		while (key>=a[i]&& i<=high);
		do
			j--;
		while(key<a[j]);
		if(i<j)
		{
		temp=a[i];
		a[i]=a[j];
		a[j]=temp;
		}
	}
	temp=a[low];
	a[low]=a[j];
```

```
        a[j]=temp;
        return j;
}
public static void qs (int a[],int low, int high)
{
        int mid;
        if(low<high)
        {
                mid=partition(a,low,high);
                qs(a,low,mid-1);
                qs(a,mid+1,high);
        }
}
public static void main(String args[])
{
        int n,i;
        FILE *f1;
        int line[50];
        int limit;
        System.out.println("Quicksort Test");
        /* Accept no.of Elements */
        System.out.println("\nEnter the number of elements");
        n=in.nextInt();
        /* create array of n elements */
        int arr[]=new int[max];
        try{
        /* Elements from file are: */
        f1= fopen(“t.txt”,”r”);
while(i<= n-1) && (fgets(line,50,f1)!=NULL))
{
fscanf(line,”%d”,&limit);
```

```
arr[i] = limit;
i++;
}
fclose(f1);
        /* Print random numbers */
        System.out.println("\nthe random elements are ");
        for(i=0;i<n;i++)
        System.out.println(arr[i]+" ");
        long start_time=System.nanoTime();
        /*call method Quick Sort*/
        qs(arr,0,n-1);
        long end_time=System.nanoTime();
        /* Print Sorted Array */
        System.out.println("\nThe Elements After sorting");
        for(i=0;i<n;i++)
        System.out.println(arr[i]+" ");
        long t=end_time - start_time;
        System.out.println("Time taken for execution is:"+t+" nanoseconds);
        }
        catch(ArrayIndexOutOfBoundsException ae)
        {
                System.out.println("Array Index reached maximum ");
        }
        }
}
```

Design and Analysis of Algorithms -21CS42- Laboratory Component

Output:

Quicksort Test

Enter the number of elements 10

the random elements are

17
31
81
44
91
24
87
42
57
17

The Elements After sorting

17
17
24
31
42
44
57
81
87
91
Time taken for execution is: 6316 nanoseconds

2. Sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator. Demonstrate using C++/Java how the divide-and-conquer method works along with its time complexity analysis: worst case, average case and best case.

Merge Sort method using divide and Conquer Technique:

Merge sort is a perfect example of a successful application of the divide and conquer technique. It sorts a given array a[0…n-1] by dividing it into two halves a[0….mid-1] and a[mid+1….high], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

Pseudocode : Mergesort(a,low , high)

```
//a[low:high] is a global array to be sorted
{
        if low<high then
        {
                //divide the problem into sub problem
                mid=(low+high)/2
                Mergesort(a,low,mid)
                Mergesort(a,mid+1,high)
                Merge(a,low,mid,high)
        }
}
```

Pseudocode : Merge(a,low,mid,high)

```
//a[low:high] is a global array containing two sorted subsets in a[low:mid] and in a[mid+1:high].
//Merge two sets into a single set in a[low:high],b[] is an auxiliary global array.
{
      h<-low
      i<-low
      j<-mid+1
      while h<=mid && j<=high do
      {
                if a[h]<=a[j] then
                {
                        b[i] <- a[h]
                        h++
                }
                else
                {
                        b[i] <- a[j]
```

```
                        j++
                }
                i++
        }
        if h>mid then

        for k=j to high do
        {
                b[i] <- a[k]
                i++
        }
        else
        for k=h to mid do
        {
                b[i] <- a[k]
                i++
        }
        for k=low to high do
                a[k] <- b[k]
}
```

Program 2

```
import java.util.Random;
import java.util.Scanner;


public class MergeSort
{
static int max=50000;
public static void mergesort(int a[],int low,int high)
{
    int mid;
    if(high>low)
    {
        mid=(low+high)/2;
        mergesort(a,low,mid);
        mergesort(a,mid+1,high);
        merge(a,low,mid,high);
    }
```

```
 }
 public static void merge(int a[],int low,int mid,int high)
 {
        int k=low,j=mid+1,i=low;
        int c[]=new int[1000];
        while((i<=mid)&&(j<=high))
        {
            if(a[i]<=a[j])
            {
                  c[k]=a[i];
                  i=i+1;
             }
            else
             {
                   c[k]=a[j];
                   j=j+1;
              }
          k=k+1;
 }
while(i<=mid)
{
     c[k]=a[i];
     k=k+1;
     i=i+1;
 }
while(j<=high)
 {
    c[k]=a[j];
    k=k+1;
    j=j+1;
}
```

```
for(i=low;i<=high;i++)
a[i]=c[i];
}
public static void main(String args[] )
{
        int n,i;
        Scanner in=new Scanner(System.in);
        Random rand=new Random();
        System.out.println("MergeSort Test");
        /* Accept no.of Elements */
        System.out.println("\nEnter the number of elements");
        n=in.nextInt();
        /* create array of n elements */
        int arr[]=new int[max];
        try{
        /* Generate Random Numbers */
        for(i=0;i<n;i++)
        arr[i]=rand.nextInt(100);
        /* Print random numbers */
        System.out.println("\nthe random elements are ");
        for(i=0;i<n;i++)
        System.out.println(arr[i]+" ");
        long start_time=System.nanoTime();
        /*call method merge Sort*/
        mergesort(arr,0,n-1);
        long end_time=System.nanoTime();
        /* Print Sorted Array */
        System.out.println("\nThe Elements After sorting");
        for(i=0;i<n;i++)
        System.out.println(arr[i]+" ");
        long t=end_time - start_time;
```

```
        System.out.println("Time taken for execution is:"+t+" nanoseconds);
        }
        catch(ArrayIndexOutOfBoundsException ae)
        {
                System.out.println("Array Index reached maximum ");
        }
        }
}
```

Output:

```
MergeSort Test
Enter the number of elements
10
the random elements are
71
66
52
97
59
48
84
32
53
54
The Elements After sorting
32
48
52
53
54
59
66
71
84
97
Time taken for execution is:21316 nanoseconds
```

Note: For n=5000 to 50000 in step of 5000 note down the time in nanoseconds. Convert time in seconds and plot the graph with n as x axis and time as y axis.

Module 3

1. Write & Execute C++/Java Program to solve Knapsack problem using Greedy method.

Fractional (Continuous) Knapsack using Greedy Method

Algorithm:
- Assume knapsack holds weight W and items have value v_i and weight w_i
- Rank items by value/weight ratio: v_i / w_i
- Thus: $v_i / w_i \geq v_{i+1} / w_{i+1}$
- Consider items in order of decreasing ratio
- Take as much of each item as possible

Pseudocode: GreedyKnapsack(m,n)

```
//p[1:n] and w[1:n] – contains the profits and weights of n objects ordered such that
//p[i]/w[i] >= p[I+1]/w[i+1]
//m is size of knapsack and x[1:n] is the solution vector
{
        for i<- 1 to n do
            x[i]=0.0
            u <- m
           for i <- 1 to n do
            {
                    if (w[i] > u) then break
                    x[i] <- 1.0
                    u <- u-w[i]
            }
       if (i<=n) then
       x[i]=u / w[i]
}
```

Program 1

```
import java.util.Scanner;
public class Knapsack
{
public static void knapsack(int n, int item[],float weight[], float profit[], float capacity)
{
                    float tp = 0,u;
                    int i;
```

```
                u =  capacity;
                float x[]=new float[20];
                for (i = 0; i < n; i++)
                x[i] = (float) 0.0;
                for (i = 0; i < n; i++)
                 {
                    if (weight[i] > u)
                      break;
                    else {
                                x[i] = (float) 1.0;
                                tp = tp + profit[i];
                                u = (int) (u - weight[i]);
                              }
                  }

                  if (i < n)
                    x[i] = u / weight[i];
                  tp = tp + (x[i] * profit[i]);
                  System.out.println("\nThe result vector is:- ");
                  for (i = 0; i < n; i++)
                    System.out.println("\tItem "+item[i]+":" +x[i]);
                  System.out.println("\nMaximum profit is:- " +tp);
}
public static void main(String[] args)
{
   float weight[]=new float[20];
   float profit[]=new float[20];
   float capacity;
   int num, i, j;
   float ratio[]=new float[20], temp;
   int item[]=new int[10];
```

```
Scanner in=new Scanner(System.in);
System.out.println("\nEnter the no. of objects:- ");
num=in.nextInt();
System.out.println("\nEnter the the items, weights and profits of each object:- ");
for (i = 0; i < num; i++)
{
    item[i]=in.nextInt();
   weight[i]=in.nextFloat();
   profit[i]=in.nextFloat();
}
System.out.println("\nEnter the capacityacity of knapsack:- ");
capacity=in.nextFloat();
for (i = 0; i < num; i++)
{
  ratio[i] = profit[i] / weight[i];
}
for (i = 0; i < num; i++)
{
   for (j = i + 1; j < num; j++)
  {
    if (ratio[i] < ratio[j])
    {
       temp = ratio[j];
       ratio[j] = ratio[i];
       ratio[i] = temp;

       temp = weight[j];
       weight[j] = weight[i];
       weight[i] = temp;

       temp = profit[j];
```

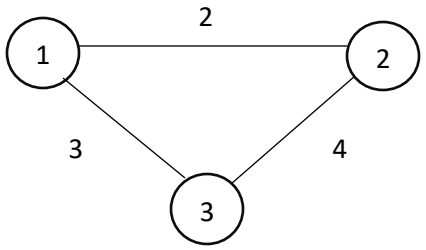
```
            profit[j] = profit[i];
            profit[i] = temp;

            temp=item[j];
            item[j]=item[i];
            item[i]=(int)temp;
          }
        }
    }
    knapsack(num, item,weight, profit, capacity);
}
}
```

Output:

```
Enter the no. of objects:-
3

Enter the items, wts and profits of each object:-
2 15 24
3 10 15
1 18 25

Enter the capacity of knapsack:-
20

The result vector is:-
        Item 2:1.0
        Item 3:0.5
        Item 1:0.0

Maximum profit is:- 31.5
```

2. Write & Execute C++/Java Program to find shortest paths to other vertices from a given vertex in a weighted connected graph, using Dijkstra's algorithm.

Single Source Shortest Paths Problem using Greedy Method :(Dijkstra's algorithm)
For a given vertex called the source in a weighted connected graph, find the shortest paths to all its other vertices. Dijkstra's algorithm is the best known algorithm for the single source shortest paths problem. This algorithm is applicable to graphs with nonnegative weights only and finds the shortest paths to a graph's vertices in order of their distance from a given source. It finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on. It is applicable to both undirected and directed graphs
Example:



Pseudocode:

```
findmin( )
{
        for i ← to n   do
                if (s[i] = 0) do
                {
                        min ← i
                        break
                }
                for i ← 1 to n do
                {
                        if (d[i]<d[min] & s[i]=0)
                        min ← i
                }
        return min
}
dijkstra( )
{
        for i ← to n   do
        {
                s[i] ← 0
                d[i] ← 999
                p[i] ← 0
        }
        d[v] ←  0
        for k ← 1 to n do
```

```
        {
                u ← findmin( )
                s[u] ← 1
                for  w1 ←1 to n do
                {
                        if (w[u][w1]!=999 & s[w1] = 0)
                        {
                                if (d[w1]>d[u]+w[u][w1] )
                                {
                                        d[w1] ← d[u]+w[u][w1]
                                        p[w1] ←u
                                }
                        }
                }
        }
        display "shortest path costs"
        for i ←1 to n do
        {
                if (d[i]=999)
                        display "sorry!no path for source v to vertex i"
                else
                        display “path cost from v to i is d[i]”
        }
        display "shortest group of paths are"
        for i ←1 to n do
        {
                if i!=v & d[i]!=999
                {
                        display i
                        j ←p[i]
                        while p[j]!=0 do
                        {
                                print "→" j
                                j ← p[j];
                        }
                        print "→"v
                }
        }
}
main( )
{
        accept number of vertices n
        accept weight matrix 999 for ∞
        accept source vertex
        call dijkstra( )
}
```

P**rogram 2**

```
import java.util.Scanner;
public class Dijkstra
{
	public static int findmin()
	{
		int i,n,min=0;
		int d[]=new int[20];
		int s[]=new int[20];
		Scanner in=new Scanner(System.in);
		n=in.nextInt();
		for(i=1;i<=n;i++)
		{
			if(s[i]==0)
			{
				min=i;
				break;
			 }
		}
		 for(i=1;i<=n;i++)
		 {
			if(d[i]<d[min] && s[i]==0)
			 min=i;
		}
		return min;
```

```
}
public void dijkstra(int v,int w[][],int s[],int d[],int n)
{
        int i,w1,u,k,j;
        int p[]=new int[20];
        for(i=1;i<=n;i++)
        {
                s[i]=0;
                d[i]=999;
                p[i]=0;
        }
        d[v]=0;
        for(k=1;k<=n;k++)
        {
                 u=findmin();
                 s[u]=1;
                for(w1=1;w1<=n;w1++)
                {
                        if(w[u][w1]!=999 && s[w1]==0)
                        {
                                if(d[w1]>d[u]+w[u][w1])
                                {
                                        d[w1]=d[u]+w[u][w1];
                                        p[w1]=u;
                                }
```

```
                }
        }
}
System.out.println("shortest path costs\n");
for(i=1;i<=n;i++)
{
if(d[i]==999)
System.out.println("sorry!  no path for source" + v + "to" + i + "vertex");
else
System.out.println("path cost from" +v+ "to" +i+ "is:" +d[i]+"\n");
}
System.out.println("shortest group of paths are\n");
for(i=1;i<=n;i++)
{
        if(i!=v && d[i]!=999)
        {
                System.out.print(i);
                 j=p[i];
                while(p[j]!=0)
                {
                        System.out.println("<----"+ j +" ");
                        j=p[j];
                }
                System.out.println("<----"+ v +"\n");
        }
```

```
            }
        }
        public static void main(String args[])
        {
             int i,j,n,v;
            int d[]=new int[20];
            int s[]=new int[20];
            int w[][]=new int[50][50];
            Dijkstra d1 = new Dijkstra();
            Scanner in=new Scanner(System.in);
            System.out.println("enter the number of vertices\n");
            n=in.nextInt();
            System.out.println("enter the cost of vertices\n");
            for(i=1;i<=n;i++)
            for(j=1;j<=n;j++)
            {
                w[i][j]=in.nextInt();
            }
            System.out.println("enter the source vertex\n");
            v=in.nextInt();
          /* call Dijkstra method */
            d1.dijkstra(v,w,s,d,n);
         }
}
```

Output:

```
enter the number of vertices
3
enter the cost of vertices
999 2 3
2 999 4
3 4 999
enter the source vertex
1
shortest path costs
path cost from1to1is:0
path cost from1to2is:2
path cost from1to3is:3
shortest group of paths are
2<----1
3<----1
```

3**. Write & Execute C++/Java Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm. Use Union-Find algorithms in your program.**

Kruskal's Algorithm using Greedy method

Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

Below are the steps for finding MST using Kruskal's algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step2 until there are (V-1) edges in the spanning tree.

The step2 uses Union-Find algorithm to detect cycle.

A disjoint-set data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union: Join two subsets into a single subset.

In this post, we will discuss an application of Disjoint Set Data Structure. The application is to check whether a given graph contains a cycle or not.

Union-Find Algorithm can be used to check whether an undirected graph contains cycle or not. This method is based on Union-Find. This method assumes that graph doesn't contain any self-loops.

Pseudocode: Kruskal(G)

```
//Kruskal's algorithm for constructing a minimum spanning tree
// Input: A weighted connected graph G=⟨V, E⟩
// Output: E_T , the set of edges composing a minimum spanning tree of G
 Sort E in non-decreasing order of the edge weights w(e_i1)<=…<=w(e_i|E|)
 E_T←∅  ; ecounter ← 0 //initialise the set of tree edges and its size
 k ← 0                 //initialise the number of processed edges
 while ecounter < |V| - 1
 k←k+1
 if E_T ∪ {e_ik}is acyclic
 E_T←E_T ∪ {e_ik}; ecounter ← ecounter+1
 Return E_T
```

Program 3

```
import java.util.Scanner;
public class Kruskal
{
	public static int find(int v,int s[])
	{
	while(s[v]!=v)
		v=s[v];
	return v;
	}
	public static void union1(int i,int j,int s[])
	{
		s[i]=j;
	}
	public static void kruskal(int n,int c[][])
	{
		int count,i,min,j,u=0,v=0,k,sum;
		int s[]= new int[10];
		int t[][]=new int[10][2];
		for(i=0;i<n;i++)
			s[i]=i;
		count=0;
```

```
sum=0;
k=0;
while(count<n-1)
{
        min=999;
        for(i=0;i<n;i++)
        {
        for(j=0;j<n;j++)
        {
                if(c[i][j]!=0 && c[i][j]<min)
                {
                        min=c[i][j];
                        u=i;
                        v=j;
                }
        }
}
if(min==999) break;
i=find(u,s);
j=find(v,s);
if(i!=j)
{
        t[k][0]=u;
        t[k][1]=v;
        k++;
```

```
                    count++;
                    sum+=min;
                    union1(i,j,s);
                }
                c[u][v]=c[v][u]=999;
        }
        if(count==n-1)
        {
                System.out.println("cost of spanning tree=" +sum+ "\n");
                System.out.println("spanning tree is\n");
                for(k=0;k<n-1;k++)
                {
                        System.out.println("\n"+t[k][0]+","+t[k][1]);
                }
        }
        System.out.println("spanning treee doesn't exist");
}
public static void main(String args[])
{
        int n,i,j;
        int c[][]=new int[10][10];
        Scanner in=new Scanner(System.in);
        System.out.println("Enter no of nodes\n");
        n=in.nextInt();
        System.out.println("Enter the cost adjacency matrix\n");
```

```
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                {
                        c[i][j]=in.nextInt();
                }
        }
        kruskal(n,c);
}
}
```

Output:

```
Enter no of nodes
6
Enter the cost adjacency matrix
999 3 999 999 6 5
3 999 1 999 999 4
999 1 999 6 999 4
999 999 6 999 8 5
6 999 999 8 999 2
5 4 4 5 2 999
cost of spanning tree=15
spanning tree is
1,2
4,5
0,1
1,5
3,5
```

4**. Write & Execute C++/Java Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.**

Prim's Algorithm using greedy Method

Prim's algorithm constructs a minimum spanning tree through a sequenceof expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set *V* of the graph's vertices. On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. (By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight. Ties can be broken arbitrarily.) The algorithm stops after all the graph's vertices have been included in the tree being constructed. Since the algorithm expands a treeby exactly one vertex on each of its iterations, the total number of such iterations is $n - 1$, where n is the number of vertices in the graph. The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.

Pseudocode:*Prim(G)*
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \{V, E\}$
//Output: *Et*, the set of edges composing a minimum spanning tree of *G*
$Vt \leftarrow \{v0\}$ //the set of tree vertices can be initialized with any vertex
$Et \leftarrow \emptyset$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
find a minimum-weight edge $e* = (v*, u*)$ among all the edges (v, u)
such that v is in $Vt$and u is in $V - Vt$
$Vt \leftarrow Vt \cup \{u*\}$
$Et \leftarrow Et \cup \{e*\}$
return *Et*

Program 4

```
import java.util.Scanner;

public class prims

{

        public static void main(String args[])

        {

                int n,i,j,min=0,a=0,u = 0,b=0,v = 0,source;

                int ne=1;

                int min_cost=0;
```

```
int cost[][]=new int[20][20];
int visited[]=new int[20];
Scanner in=new Scanner(System.in);
System.out.println("Enter the no. of nodes:");
n=in.nextInt();
System.out.println("Enter the cost matrix:\n");
for(i=1;i<=n;i++)
{
        for(j=1;j<=n;j++)
        {
                cost[i][j]=in.nextInt();
        }
}
for(i=1;i<=n;i++)
        visited[i]=0;
System.out.println("Enter the root node:");
source=in.nextInt();
visited[source]=1;
System.out.println("\nMinimum cost spanning tree is\n");
while(ne<n)
{
        min=999; for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                {
```

```
                                        if(cost[i][j]<min) if(visited[i]==0)
                                                continue;
                                        else
                                        {
                                                min=cost[i][j];
                                                a=u=i;
                                                b=v=j;
                                        }
                                }
                        }
 if(visited[u]==0||visited[v]==0)
 {
        ne++;
        System.out.println("\nEdge" + ne + "\t" +a+ "->" +b+ "=" +min+"\n");
        min_cost=min_cost+min;
        visited[b]=1;
 }
 cost[a][b]=cost[b][a]=999;
 }
 System.out.println("\nMinimum cost="+min_cost+"\n");
 }
}
```

Output:

```
Enter the no. of nodes:
4
Enter the cost matrix:
```

```
999 1 5 2
1 999 999 999
5 999 999 3
2 999 3 999
Enter the root node:
1
Minimum cost spanning tree is
Edge2  1->2=1
Edge3  1->4=2
Edge4  4->3=3
Minimum cost=6
```

Design and Analysis of Algorithms -21CS42- Laboratory Component

Module 4

1) Write C++/ Java programs to solve All-Pairs Shortest Paths problem using Floyd's algorithm.

.

Floyd's Algorithm for All-Pairs Shortest-Paths Problem using Dynamic Programming

Given a weighted connected graph (undirected or directed), the ***all-pairs shortest paths problem*** asks to find the distances—i.e., the lengths of the shortest paths—from each vertex to all other vertices. This is one of several variations of the problem involving shortest paths in graphs. Because of its important applications to communications, transportation networks, and operations research, it has been thoroughly studied over the years. Among recent applications of the all-pairs shortest-path problem is precomputing distances for motion planning in computer games.

It is convenient to record the lengths of shortest paths in an $n \times n$ matrix D called the ***distance matrix***: the element dij in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex. We can generate the distance matrix with an algorithm that is very similar toWarshall's algorithm. It is called ***Floyd's algorithm*** after its co-inventor Robert W. Floyd. It is applicable to both undirected and directed weighted graphs provided that they do not contain a cycle of a negative length. (The distance between any two vertices in such a cycle can be made arbitrarily small by repeating the cycle enough times.) The algorithm can be enhanced to find not only the lengths of the shortest paths for all vertex pairs but also the shortest paths themselves.

Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of $n \times n$ matrices:
$D(0), \ldots, D(k-1), D(k), \ldots, D(n)$.

Pseudocode:*Floyd(W*[1..*n,* 1..*n*]*)*
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix *W* of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
$D \leftarrow W$ //is not necessary if *W* can be overwritten
for $k \leftarrow 1$ **to** n **do**
for $i \leftarrow 1$ **to** n **do**
for $j \leftarrow 1$ **to** n **do**
$D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
return *D*

Program 1

```
import java.util.Scanner;

public class Floyds {

        public static void floyd(int a[][],int n)
```

```
{
        int i,j,k;
        int d[][]=new int[10][10];
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                {
                        d[i][j]=a[i][j];
                }
        }
        for(k=1;k<=n;k++)
        {
                for(i=1;i<=n;i++)
                {
                        for(j=1;j<=n;j++)
                        {
                                d[i][j]=min(d[i][j],d[i][k]+d[k][j]);
                        }
                }
        }
System.out.println("\nThe distance matrix is\n");
for(i=1;i<=n;i++)
{
        for(j=1;j<=n;j++)
        {
```

```
                System.out.print(+d[i][j]+"\t");
            }
            System.out.println("\n");
        }
 }
public static int min (int a,int b)
{
        if(a<b)
            return a;
        else
            return b;
}
public static void main(String args[])
{
         int n,i,j;
         int a[][]=new int[10][10];
         Scanner sc=new Scanner(System.in);
        System.out.println("Enter the no.of nodes : ");
        n=sc.nextInt();
        System.out.println("\nEnter the cost adjacency matrix");
        for(i=1;i<=n;i++)
         for(j=1;j<=n;j++)
        a[i][j]= sc.nextInt();
        floyd(a,n);
}
}
```

Design and Analysis of Algorithms -21CS42- Laboratory Component

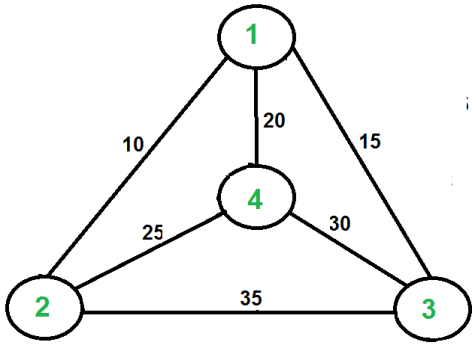
Output:

```
Enter the no.of nodes :
4
Enter the cost adjacency matrix
0 999 3 999
2 0 999 999
999 7 0 1
6 999 999 0
The distance matrix is
0       10      3       4
2       0       5       6
7       7       0       1
6       16      9       0
```

2) Write C++/ Java programs to solve Travelling Sales Person problem using Dynamic programming.

Travelling Salesman Problem(TSP) using Dynamic Programming

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.The problem is a famous NP hard problem. There is no polynomial time know solution for this problem.

Following are different solutions for the travelling salesman problem.

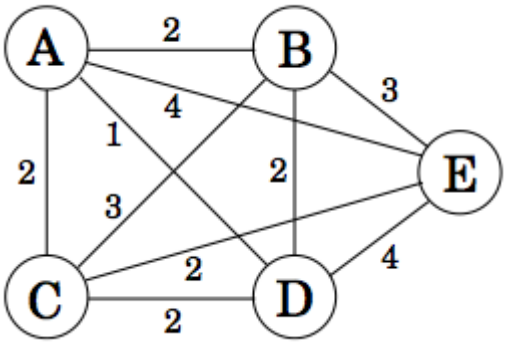
Naive Solution:

1) Consider city 1 as the starting and ending point.
2) Generate all (n-1)! Permutations of cities.
3) Calculate cost of every permutation and keep track of minimum cost permutation.
4) Return the permutation with minimum cost.

Time Complexity: (n!)

Dynamic Programming:

Denote the cities by 1,...,n, the starting city being 1, and let D = (*dij*) be the matrix of intercity distances. The goal is to design a tour that starts and ends at 1, includes all other cities exactly once, and has minimum total length. Figure shows an example involving five cities.



Let's dive right into the DP. So what is the appropriate sub-problem for the TSP? In this case the most obvious partial solution is the initial portion of a tour. Suppose we have started at city 1 as required, have visited a few cities, and are now in city j. What information do we need in order to extend this partial tour? We certainly need to know j, since this will determine which cities are most convenient to visit next. And we also need to know all the cities visited so far, so that we don't repeat any of them. Here, then, is an appropriate sub-problem.

For a subset of cities $S \subseteq \{1,2,...,n\}$ that includes 1, and $j \in S$, let C(S,j) be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j. When $|S| > 1$, we define $C(S, 1) = \infty$ since the path cannot both start and end at 1. Now, let's express C(S,j) in terms of smaller sub-problems. We need to start at 1 and end at j; what should we pick as the second-to-last city? It has to be some $i \in S$, so the overall path length is the distance from 1 to i, namely, $C(S - \{j\}, i)$, plus the length of the final edge, d_{ij}.
We must pick the best such i:

$$C(S,j) = \min_{i \in S, i \neq j} C(S - \{j\}, i) + d_{ij}$$

The sub-problems are ordered by |S|. Here's the code.

```
C({1},1) = 0
for s = 2 to n:
   for all subsets S ⊆ {1,2,...,n} of size s and containing 1:
      C(S,1) = ∞
      for all j∈S,j≠1:
         C(S, j) = min{C(S−{j},i)+dij:i∈S,i≠j}
return minjC({1,...,n},j)+dj1
```

There are at most $2n.n$ sub-problems, and each one takes linear time to solve.
The total running time is therefore $O(n^2.2^n)$

Program 2

```
import java.util.Scanner;

public class Tsp
```

```
{
        final static  int MAX=100;
        final static  int INFINITY=999;
        static int tsp_dp(int c[][],int tour[],int start,int n)
        {
                int i,j,k;
                int temp[]=new int[MAX];
                int mintour[]=new int[MAX];
                int mincost,ccost;
                if(start==n-2)
                        return(c[tour[n-2]][tour[n-1]]+c[tour[n-1]][0]);
                mincost=INFINITY;
                for(i=start+1;i<n;i++)
                {
                        for(j=0;j<n;j++)
                                temp[j]=tour[j];
                                temp[start+1]=tour[i];
                                temp[i]=tour[start+1];
                        if(c[tour[start]][tour[i]]+(ccost=tsp_dp(c,temp,start+1,n))<mincost)
                        {
                                mincost=ccost+c[tour[start]][tour[i]];
                                for(k=0;k<n;k++)
                                        mintour[k]=temp[k];
                        }
                }
```

```
                for(i=0;i<n;i++)
                        tour[i]=mintour[i];
                tour[i]=start;
                return mincost;
 }
public static void main(String[] args)
{
   int n,i,j,cost;
   int c[][]=new int[MAX][MAX];
   int tour[]=new int[MAX];
   Scanner sc=new Scanner(System.in);
   System.out.println("Enter the number of cities:");
   n=sc.nextInt();
   System.out.println("Enter the cost matrix\n");
   for(i=0;i<n;i++)
   for(j=0;j<n;j++)
            c[i][j]=sc.nextInt();
   for(i=0;i<n;i++)
            tour[i]=i;
   cost=tsp_dp(c,tour,0,n);
   System.out.println("\nmincost by dp:"+cost);
   System.out.println("\ntour:  ");
   for(i=0;i<n;i++)
   System.out.print(tour[i]+1 +"\t");
}
```

}

Output:

```
Enter the number of cities:
5
Enter the cost matrix
0 3 1 5 8
3 0 6 7 9
1 6 0 4 2
5 7 4 0 3
8 9 2 3 0
mincost by dp:16
tour:
1       2       4       5       3
```

3. Write C++/ Java programs to solve 0/1 Knapsack problem using Dynamic Programming method.

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

Optimal Substructure:
To consider all subsets of items, there can be two cases for every item:

(1) the item is included in the optimal subset, (2) not included in the optimal set.
Therefore, the maximum value that can be obtained from n items is max of following two values.
1) Maximum value obtained by n-1 items and W weight (excluding nth item).
2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

0/1(Discrete) Knapsack Pseudocode:

```
max(a,b)
{
return(a>b)?a:b;
}

 knap(i, j)
 {
        if(i=0 or j=0) then
                        v[i][j]=0
                elseif(j<w[i]) then
                        v[i][j]=knap(i-1,j)
                else
                        v[i][j]=max(knap(i-1,j), value[i]+knap(i-1,j-w[i]))
                returnv[i][j]
  }
 optimal( i,j)
  {
                if(i>=1 or j>=1) then
```

```
                    if(v[i][j]!=v[i-1][j]) then
                    {
                        Print Item i
                        b[i]=1
                        j=j-w[i]
                        optimal(i-1,j)
                    }
                    else
                    optimal(i-1,j);
  }
```

Program 3

```
import java.util.Scanner;
public class Knapsack1
{
    private static int w[]=new int[10];
    private static int b[]=new int[10];
    private static int v[][]=new int[10][10];
    private static int value[]=new int[10];
    static int max(int a, int b)
    {
        return(a>b)?a:b;
    }
    static int knap(int i,int j)
    {
        if(i==0 || j==0)
            v[i][j]=0;
        else if(j<w[i])
            v[i][j]=knap(i-1,j);
        else
            v[i][j]=max(knap(i-1,j), value[i]+knap(i-1,j-w[i]));
        return v[i][j];
    }
    static void optimal(int i,int j)
```

```
{
        if(i>=1 || j>=1)
                if(v[i][j]!=v[i-1][j])
                {
                        System.out.println("Item:"+i);
                        b[i]=1;
                        j=j-w[i];
                        optimal(i-1,j);
                }
                else
                        optimal(i-1,j);
}
public static void main(String[] args)
{
        int profit, w1,n,i,j;
        Scanner sc=new Scanner(System.in);
        System.out.println("enter the number of items:");
        n=sc.nextInt();
        System.out.println("enter the capacity of the knapsack:");
        w1=sc.nextInt();
        System.out.println("enter the values:");
        for(i=1;i<=n;i++)
        value[i]=sc.nextInt();
        System.out.println("enter the weights:");
        for(i=1;i<=n;i++)
        w[i]=sc.nextInt();
        profit=knap(n,w1);
        System.out.println("profit: "+profit);
        System.out.println("\n optimal subset is:\n");
        optimal(n,w1);
        System.out.println("the solution vector is:");
```

```
            for(i=1;i<=n;i++)
            System.out.println(b[i]);
        }
}
```

Output:

```
enter the number of items:
4
enter the capacity of the knapsack:
2
enter the values:
3
45
4
3
enter the weights:
1
1
1
1
profit: 49

optimal subset is:

Item:3
Item:2
the solution vector is:
0
1
1
0
```

Module 5

1. Design and implement C++/Java Program to find a subset of a given set S = {Sl, S2,…, Sn} of n positive integers whose SUM is equal to a given positive integer d. For example, if S = {1, 2, 5, 6, 8} and d= 9, there are two solutions {1, 2, 6} and {1, 8}. Display a suitable message, if the given problem instance doesn't have a solution.

Algorithm for Subset

Algorithm for main:
Step 1: accept number of elements
Step 2: accept set of elements
Step 3: accept maximum subset value
Step 4: print the subset cell sum of sub(0,1,sum)
Step 5: stop

Pseudocode:

```
sumofsub(0,1,sum)
{
        x[k]=1
        if(s +w[k] <- u)
        {
                print solution v++
                for i<- 1 to n do
                        if x[i]=1
                                print w[i]
        }
        else if (s+ w[k] +w[k+] < = m) do
                call sumofsub(s + w[k],k+1 , r-w[k])
        if( s +r-w[k]>=m and s+w[k+] <=m) do
        {
                x[k]<-0
                call sumofsub(s,k+1,r-w[k])
        }
}
```

Program 1

```
import java.util.Scanner;

public class Subset

{

        private static  int d;

        private static  int count=0;
```

```
private static int x[]=new int[20];
private static int w[]=new int[20];
public static void main(String[] args)
{
        Scanner sc=new Scanner(System.in);
        int i,n,sum=0;
        System.out.println("Enter the no. of elements: ");
        n=sc.nextInt();
        System.out.println("\nEnter the elements in ascending order:\n");
        for(i=0;i<n;i++)
        w[i]=sc.nextInt();
        System.out.println("\nEnter the sum: ");
        d=sc.nextInt();
        for(i=0;i<n;i++)
        sum=sum+w[i];
        if(sum<d)
        {
                System.out.println("No solution\n");
                return;
        }
        subset(0,0,sum);
        if(count==0)
        {
                System.out.println("No solution\n");
                return;
        }
}
static void subset(int cs,int k,int r)
{
        int i;
        x[k]=1;
```

```
            if(cs+w[k]==d)
            {
                System.out.println("\n\nSubset" +(++count));
                for(i=0;i<=k;i++)
                if(x[i]==1)
                System.out.println(w[i]+" ");
            }
            else if(cs+w[k]+w[k+1]<=d)
            {
                subset((cs+w[k]),k+1,r-w[k]);
            }
            if(cs+r-w[k]>=d && cs+w[k]<=d)
            {
                x[k]=0;
                subset(cs,k+1,r-w[k]);
            }
        }
}
```

Output:

```
Enter the no. of elements:
5
Enter the elements in ascending order:
1 2 5 6 8
Enter the sum:
9
Subset1
1
2
6
Subset2
1
8
```

2. Design and implement C++/Java Program to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using backtracking principle.

Hamiltonian Path Problem using Backtracking Method

Hamiltonian Path is a path in a directed or undirected graph that visits each vertex exactly once. The problem to check whether a graph (directed or undirected) contains a Hamiltonian Path is NP-complete, so is the problem of finding all the Hamiltonian Paths in a graph. Following images explains the idea behind Hamiltonian Path more clearly.

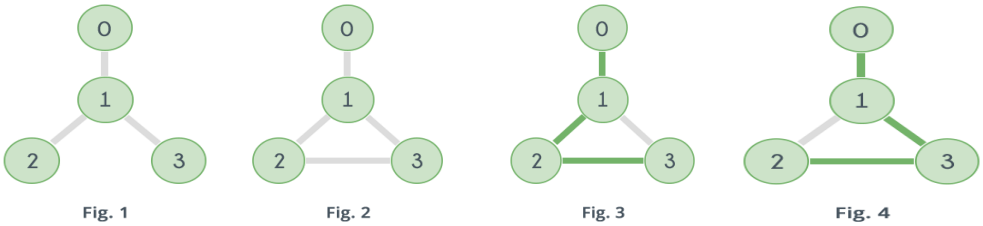


Fig. 1 Fig. 2 Fig. 3 Fig. 4

Graph shown in Fig.1 does not contain any Hamiltonian Path. Graph shown in Fig. 2 contains two Hamiltonian Paths which are highlighted in Fig. 3 and Fig. 4

Following are some ways of checking whether a graph contains a Hamiltonian Path or not.

1. A Hamiltonian Path in a graph having N vertices is nothing but a permutation of the vertices of the graph [v_1, v_2, v_3, ...v_{N-1}, v_N] , such that there is an edge between v_i and v_{i+1} where $1 \leq i \leq N-1$. So it can be checked for all permutations of the vertices whether any of them represents a Hamiltonian Path or not. For example, for the graph given in Fig. 2 there are 4 vertices, which means total 24 possible permutations, out of which only following represents a Hamiltonian Path.
   0-1-2-3
   3-2-1-0
   0-1-3-2
   2-3-1-0

2. **Pseudocode:**

```
function check_all_permutations(adj[][], n)
{
  for i = 0 to n
     p[i]=i
  while next permutation is possible
     valid = true
     for i = 0 to n-1
        if adj[p[i]][p[i+1]] == false
           valid = false
           break
     if valid == true
        return true
     p = get_next_permutation(p)
  return false
```

```
    }
```

The function get_next_permutation(p) generates the lexicographically next greater permutation than p.

Following is the C++ implementation:

```
    bool check_all_permutations(bool adj[][MAXN], int n){
        vector<int>v;
        for(int i=0; i<n; i++)
            v.push_back(i);
        do{
            bool valid=true;
            for(int i=0; i<v.size()-1; i++){
                if(adj[v[i]][v[i+1]] == false){
                    valid=false;
                    break;
              }
               }
               if(valid)
                   return true;
        }while(next_permutation(v.begin(), v.end()));
        return false;
    }
```

Time complexity of the above method can be easily derived. For a graph having N vertices it visits all the permutations of the vertices, i.e. N! iterations and in each of those iterations it traverses the permutation to see if adjacent vertices are connected or not i.e N iterations, so the complexity is O(N * N!).

Program 2

```
public class Hamiltonian
{
   final int V = 5;
   int path[];


   /* A utility function to check if the vertex v can be added at index 'pos'in the Hamiltonian
      Cycle constructed so far (stored in 'path[]') */


   boolean isSafe(int v, int graph[][], int path[], int pos)
   {
      /* Check if this vertex is an adjacent vertex of the previously added vertex. */
          if (graph[path[pos - 1]][v] == 0)
                  return false;
```

```
    /* Check if the vertex has already been included.This step can be optimized by creating
    an arrayof size V */
        for (int i = 0; i < pos; i++)
        if (path[i] == v)
        return false;
    return true;
  }
  /* A recursive utility function to solve hamiltonian cycle problem */
  boolean hamCycleUtil(int graph[][], int path[], int pos)
  {
    /* base case: If all vertices are included in Hamiltonian Cycle */
        if (pos == V)
        {
        // And if there is an edge from the last included
        // vertex to the first vertex
        if (graph[path[pos - 1]][path[0]] == 1)
                return true;
        else
                return false;
        }
    /* Try different vertices as a next candidate in Hamiltonian Cycle. We don't try for 0 as
      We included 0 as starting point in hamCycle() */
    for (int v = 1; v < V; v++)
    {
        /* Check if this vertex can be added to Hamiltonian Cycle */
          if (isSafe(v, graph, path, pos))
{
          path[pos] = v;
            /* recur to construct rest of the path */
        if (hamCycleUtil(graph, path, pos + 1) == true)
```

```
                return true;
            /* If adding vertex v doesn't lead to a solution,then remove it */
            path[pos] = -1;
        }
    }
    /* If no vertex can be added to Hamiltonian Cycle constructed so far, then return false */
    return false;
}
/* This function solves the Hamiltonian Cycle problem using Backtracking. It mainly uses
 hamCycleUtil() to solve the problem. It returns false if there is no Hamiltonian Cycle
possible, otherwise return true and prints the path.Please note that there may be more
  than one solutions this function prints one of the feasible solutions. */

 int hamCycle(int graph[][])
{
    path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    /* Let us put vertex 0 as the first vertex in the path.If there is a Hamiltonian Cycle,
 then the path can be started from any point of the cycle as the graph is undirected */

    path[0] = 0;
    if (hamCycleUtil(graph, path, 1) == false)
    {
        //System.out.println("\nSolution does not exist");
        return 0;
    }
    printSolution(path);
    return 1;
}
```

```
/* A utility function to print solution */
void printSolution(int path[])
{
    System.out.println("Solution Exists: Following" +" is one Hamiltonian Cycle");
    for (int i = 0; i < V; i++)
        System.out.print(" " + path[i] + " ");

    // Let us print the first vertex again to show the
    // complete cycle
    System.out.println(" " + path[0] + " ");
}
// driver program to test above function
public static void main(String args[])
{
    Hamiltonian hamiltonian =new Hamiltonian();
    /* Let us create the following graph
      (0)--(1)--(2)
       |  / \  |
       | /   \ |
       |/     \|
      (3)-------(4)    */
    int graph1[][] = {{0, 1, 0, 1, 0},
       {1, 0, 1, 1, 1},
       {0, 1, 0, 0, 1},
       {1, 1, 0, 0, 1},
       {0, 1, 1, 1, 0},
    };
    // Print the solution
 hamiltonian.hamCycle(graph1);
    /* Let us create the following graph
```

```
        (0)--(1)--(2)
         |   / \   |
         |  /   \  |
         | /     \ |
        (3)       (4)    */
    int graph2[][] = {{0, 1, 0, 1, 0},
       {1, 0, 1, 1, 1},
       {0, 1, 0, 0, 1},
       {1, 1, 0, 0, 0},
       {0, 1, 1, 0, 0},
    };
    // Print the solution
    hamiltonian.hamCycle(graph2);
  }
}
```

Output:

```
Solution Exists: Following is one Hamiltonian Cycle
 0  1  2  4  3  0
```