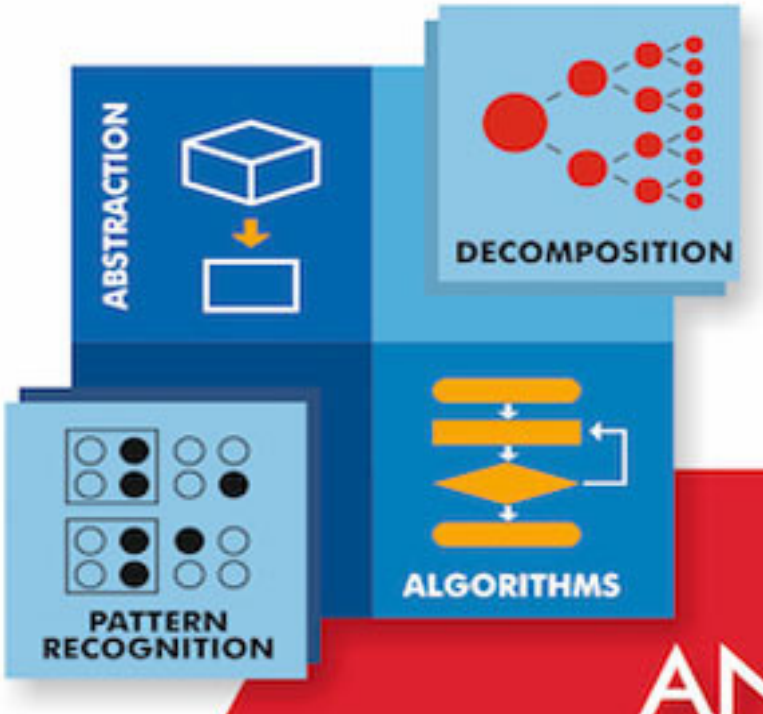


A process or set of rules to be followed in **CALCULATIONS OR OTHER PROBLEM-SOLVING OPERATIONS,** especially by a computer.

ANALYSIS AND DESIGN OF ALGORITHMS

A BEGINNER'S HOPE...

- Algorithm & Algorithmic Strategy
- Complexity of Algorithms
- Divide-and-Conquer Algorithms
- Greedy Algorithm
- Dynamic Programming
- Graph Theory
- Backtracking Algorithms
- Branch and Bound Algorithms
- String-Matching Algorithms
- P and NP Problems

SHEFALI SINGHAL
NEHA GARG

BPB PUBLICATIONS

Analysis and Design

of Algorithms

A Beginner's Hope

By

SHEFALI SINGHAL

NEHA GARG

FIRST EDITION 2018

ISBN: 978-93-8655-189-4

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

Ph: 24756967/24756400

Published by Manish Jain for BPB Publications, 20, Ansari Road, Darya Ganj, New Delhi-110002 and Printed by Repro India Pvt Ltd, Mumbai

PREFACE

As we are moving forward in this current computer era, we emphasize on those techniques which are more efficient and economic. This book offers a variety of algorithm's design technique, their complexity, efficiency and many more aspects. It also focuses on algorithms for games and for various machines used by us in our daily life.

Keeping in mind the changing scenario in software development, Authors are confident to explain algorithmic techniques which may prove as a first step for a beginner.

This book is written as per the syllabus of Diploma in Computer Science, B.Tech./B.E., BCA, Bsc., M.Tech./M.E., MCA, Msc.

The book is organized in 10 chapters. Chapter 1 & 2 provides the basics of algorithms and their complexity. Chapter 3 contains divide and conquer techniques. Chapter 4 & 5 explains greedy and dynamic approach with basic comparisons between them. Graph theory is explained in chapter 6 providing a basis to use dynamic as well as greedy algorithms for various kinds of trees. Chapter 7 & 8 discuss about backtracking, branch and bound techniques which deal with problems with tree data structure. Then we talk about string matching algorithms in Chapter 9 . Lastly, chapter 10 covers NP problems.

All the basic concepts have been discussed in above chapters, however, for practice session students may follow chapter wise review exercises and the set of question papers in Appendix.

Although the content has been supervised by many senior educationists, there may be some shortcomings. As human is prone to errors, authors wish to forgive the mistakes and we welcome your suggestions and criticism if any! The authors will try their best to incorporate such valuable suggestions in the subsequent editions of this book.

DEDICATED TO

This book is dedicated to our dearest parents and beloved husband who ever and ever supported us in all respects of life and career. This journey of ours proved to be a boon by following their words and experiences.

“Behind every young child who believes in himself is a Parent who Believed First”

Mathew Jacobson

ACKNOWLEDGEMENT

First and foremost, we would like to thank God as we could never have done this without the faith we have in the Almighty.

We would like to express our gratitude to the Computer Science & Engineering Department of Manav Rachna International University for always being good to us.

We would extend our gratitude to the BPB Publication for bringing out the book in its present form.

A great thanks to our family and friends, without their support we wouldn't be able to complete this task.

“Parents are always been the boost up part behind a rising sun”.

Shefali Singhal

Neha Garg

Table of Contents

C HAPTER -1

Algorithm & Algorithmic Strategy

In this chapter student will understand:

What is an Algorithm? From where this term is originated? Why should one study algorithm? What is the role of an algorithm in other fields? What will happen if algorithms vanish from the picture?

1.1 Algorithm

Given a particular problem, how one can solve it on the computer?

The way you solve any problem in systematic manner by using some input, by following proper sequence of steps and finally getting the desired output. Hence we may say that solving a problem is just like following a recipe to cook a dish. As we can see that a recipe is a box of steps and following those steps in proper sequence will always give the output.

Once we design any algorithm, we need to know how well it works for a given input. Is there any algorithm better than this? In order to answer much such type of queries, algorithm analysis came into the picture.

1.2 History

The term algorithm was named after the name of a Persian author, Abu Ja'far Muhammad IbnMusa'al Khowarizmi (c. 825 A. D.), who has given the definition of the algorithm as:

An algorithm is a set of rules for carrying out computation either by hand or by some machine.

It is a set of procedures that use some input and give desired output. He said that an algorithm must satisfy the following properties:

Input: finite no of input must be externally supplied.

Output: At least one output must be produced.

Definiteness: Each instruction must be clear with respect to upper bound.

Effectiveness: Each instruction must have a meaningful result after execution.

Finiteness: If each instruction is executed in a proper sequence then the process must successfully terminate in finite time without going into any infinite loop.

1.3 Scope of Algorithms in different fields

Practically every task that you perform using a computer system depends indirectly on an algorithm that someone has worked previously very hard to make sense of it. Indeed, even the simplest application on a cutting edge computer would not be conceivable to be utilized without calculations. Algorithms are being utilized to oversee memory and load information from the hard drive.

Obviously, there are circumstances when you'll come across a problem that has not been previously studied. In these cases, you have to come up with a new algorithm or apply an old algorithm in a new way. More you think about algorithms in this case better is the odds of discovering rationale to take care of the issue. In many cases, a new problem can be reduced to an old problem without excessive amount of effort, but you need to have a basic fundamental understanding of the previous problem in order to solve the current one.

As an example of this, let's consider what a switch does on the Internet. A switch has N cables plugged into it and gets bundles of information rolling in from the links. The switch has to first analyze the packets and then send them back to the correct cables. A switch, similar to a PC, is controlled by a clock with discrete strides - the packets are sending out at discrete intervals, rather than continuously. In a fast switch, we want to send out as many packets as possible during each interval, so they don't stack up and get dropped. The objective of the algorithm we need to create is to convey how many packets as could be expected under the circumstances during each interval, and furthermore also to send them out with the goal that the ones that arrived earlier get sent out earlier. In this case, it turns out that an algorithm for a problem that is known as “stable matching” is specifically material to our concern, though at first glance this relationship appears to be far-fetched. Only through pre-existing algorithmic knowledge and understanding such a relationship can be found.

Algorithms have a vital role in different areas, few of which we have discussed below

Data Structures is the real area for existing algorithms and also for generation of new algorithms.

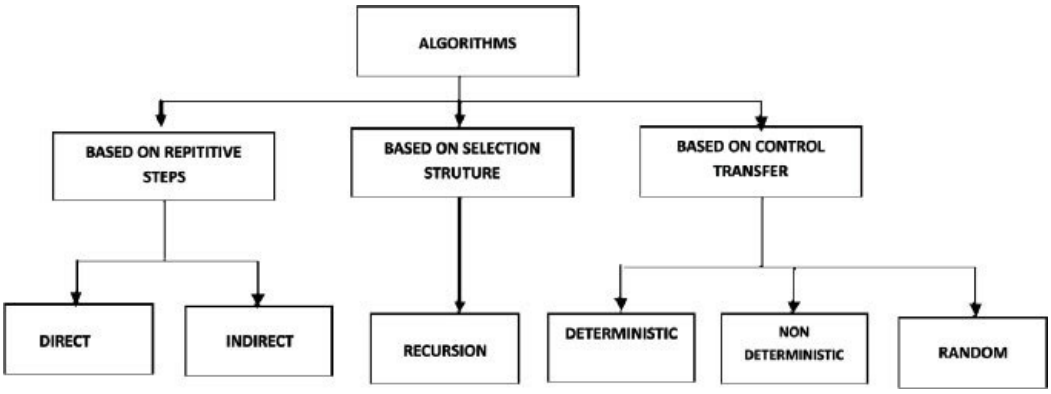
Cryptography is another study area in computer science where all the techniques like RSA, AES, DES, etc. are completely based on the algorithm.

Manufacturing units fully utilize automation. There an algorithms who plays an important role.

In biological field, things like a number of genes in DNA, ECG coding algorithm, etc.

In the electrical field like embedded programming for IC's, magnetic resonance imaging (MRI), etc.

1.4 Classification Of Algorithms



The classification of an algorithm is based on repetitive execution of the steps and control transfer from one statement to another.

By repetitive steps, an algorithm can be further classified into two types.

Direct Algorithm : In this algorithm, one should know the number of iterations in advance.

For example, to display numbers from 1 to 10, the loop variable should be initialized from 1 to 10. The statement would be as follows

for (j=1;j<=10;j++)

In the above statement, it is observed that the loop will iterate ten times.

Indirect Algorithm : In this type of algorithm, repetitively steps are executed. It is totally unknown that how many repetitions are to be made.

For example, the repetitive steps are as follows.

To find the first five Armstrong numbers from 1 to n, where n is the fifth Armstrong number.

To find the first three palindrome numbers.

Algorithm on the basis of selection structure

Recursive Algorithm : This type of algorithm solves the base cases directly, recurs with a simpler sub problem and does some extra work to convert the solution to the simpler sub problem that produces a solution to the given problem.

Example :

To count the number of elements in a list.

Tower of Hanoi problem

Note: A recursive method is a method that calls itself either directly or indirectly. Each repetition of the process is called iteration and the result of iteration is used as the starting point for the next iteration.

Based on the control transfer, there are three categories of algorithms which are discussed below:

Deterministic: A Deterministic algorithm is either follows a ‘yes’ path or ‘no’ path based on the condition. In these algorithms, when control reaches for decision logic, two paths ‘yes and ‘no’ are shown. Now the program control follows one of the routes depending upon the given condition.

Example:

Testing whether a number is even or odd.

Testing whether a number is positive or negative.

Non-Deterministic: In this type of algorithm to reach to the solution, we have one of the multiple paths.

Example:

To find a day of a week.

To find a path between two stations like, path from Delhi to Chennai.

Random algorithm: Random algorithms are algorithms that make some spontaneous decision during their execution. After executing few instructions the control of the program transfers to another, randomly.

Example:

A random search

Random Key generator in cryptography.

1.5 Pseudo Code

In computer theory, we distinguish between algorithm and computer program. Algorithms satisfy the property of finiteness while the program may run in an infinite loop due to waiting for some resource to be allocated, couldn't find terminating condition, etc.

Pseudo code is the informal and compact representation of computer programming algorithm that uses conventions of the structured programming languages like C, C++, etc. It is only intended for human reading rather than machine reading, that's why we write pseudo code in the natural language. There is one more method to represent algorithm in the symbolic representation called flowchart, but we can draw them only for simpler and smaller algorithms.

Pseudo-code typically hides details that are not much essential for understanding the algorithm or may create complexity, such as functions (or subroutines), variable declaration, semicolons, special words and so on. Any version of pseudo-code is acceptable as long as its instructions are unambiguous. Pseudo-code is independent of any programming language. It cannot be compiled or executed, and do not follow any syntax rules.

Pseudo Code Convention

Give an appropriate name to your pseudo code and your variables. An identifier must begin with a letter.

The steps are numbered. Subordinate numbers and/or indentation must use for dependent statements in selection and repetition structures.

Use “//” for comments.

For input use word READ . There is no need to declare variable type.

For output use word DISPLAY < variable name>.

For simple computation use word COMPUTE expression

Example : Write Pseudo code to add two numbers

Read a, b

Compute c as sum of a and b

Display c.

For assignment use “←.”

Selection

Single-Selection IF

IF condition THEN (IF condition is true, then pass control to subordinate statements of 1, etc. If condition is false, then skip statements)

1.1 Statement 1

1.2 Etc.

Double-Selection IF and ELSE

IF condition THEN (IF condition is true, then execute subordinate statement of 1, etc. If condition is false, then skip statements and execute statements under ELSE)

1.1 statements 1

1.2 etc.

ELSE (else if condition is not true, then do subordinate statement of 2, etc.)

2.1 statement 2

2.2 statement 3

END IF

Example: Write Pseudo code to find whether a given number is even or odd.

Read a

Compute modulus of a and 2

If the modulus of a and 2 is zero Then

i. Display a is even

Else

i. Display a is odd

end if

Switch case

Switch (expression) TO

(1) case 1: action1

(2) case 2: action2

(3) etc.

(4) default: action x

Repetition

WHILE condition (while condition is true, then execute subordinate statements) do

While(condition) do

1.1 statement 1

1.2 etc.

Example: Write Pseudo code to print numbers from 1 to 10

while i<10 do

i. Display i

ii. i← i +1

end while.

Do (DO - WHILE structure like WHILE, but tests condition at the end of the loop. Thus, the instructions in Do-block will be executed once compulsorily.) do

2.1 statement 1

2.2 etc.

while condition

Example: Write Pseudo code to print numbers from 10 to 1

I do

(i) Display A

(ii) A ← A - 1

ll while A> 0

For upper and lower bounds of repetition

For (intial;condition;increment/decrement)

3.1 statements 1

3.2 etc.

Array elements are represented by specifying the array name followed by the index in square brackets. For example, indicates the i th element of the array A.

Example: Pseudo code for inserting ten elements in an array A[10] of size 10.

For j ← 1 to 10 do

1.1 Read A[j].

1.2 j ← j + 1.

end for.

1.6 How to design an algorithm?

In computer science, designing an algorithm is an art or skill. Before actual implementation of a program, designing an algorithm is a necessary step.

Suppose we want to build a house we do not directly start constructing the house. Instead, we can consult an architect, we put our ideas and suggestion, the architect note it down and makes changes in the plan accordingly. This process continues till we are satisfied. Finally, the blue print of house get ready. Once the design process is over actual construction of building will start. Now it becomes very easy and systematic for the construction of desired house. In this example, you will find out that all designing is just a paper work and at that instance, if we want some changes then those can be easily carried out on paper. After a satisfactory design, the construction activities start. Same is a program development process.

Let us list " what are the steps to be followed while designing and analyzing an algorithm "?

These steps are as follows:

Understanding the problem

Decision making on

(a) Capabilities of computational devices

(b) Choosing either exact or approximate method for problem-solving

(c) Data structures

(d) Algorithmic strategies

Specification of algorithm

Algorithmic verification

Analysis of algorithm

Implementation or coding of algorithm

Fig.: Steps in Analysis and Design of Algorithm

Let us now discuss each step in detail:

1. Understanding the problem

In this step, first of all, you need to figure out the problem statement completely. While realizing the problem statement, read the problem description carefully and ask questions for clarifying the doubts about the problem.

After understanding the problem statements find out what are the necessary inputs for solving that problem. The input to the problem is called the instance of the problem. It is essential to decide the range of inputs so that the boundary values of the algorithm get fixed. The algorithm should work correctly for all valid inputs.

1. Decision making

After finding the required input set for the given problem we have to analyze the input and need to decide certain issues:

Capabilities of computational devices

Globally we can classify an algorithm from the execution point of view as:

Sequential Algorithm

It mainly runs on the machine in which the instructions are executed one after the other. Such a system is called as random access machine (RAM).

Parallel Algorithm

These algorithms run on the machine which executes the instructions in parallel.

Other than this, there are certain complex problems which require a huge amount of memory or the problems for which execution time is a major factor. For solving such problems, it is essential to have a proper choice of the computational device which is space and time efficient.

Choice for either exact or approximate method for problem-solving

If the problem needs to be computed correctly, then we need the exact algorithm . Otherwise, if the problem is too complex that we won't get the proper solution then in such cases we usually choose approximation algorithm .

For example, traveling salesman problem follows the approximation algorithm.

Data structures

Data structure and algorithm work together as they are interdependent. Hence the choice of proper data structure is required before designing the actual algorithm. To implement the designed algorithm, i.e., for programming, we need the data structure.

Algorithmic strategies

An algorithmic strategy is a general approach by which numerous issues can be resolved. Sometimes it is called as algorithmic techniques because algorithms are categorized based on areas of computing. Here we strategize the best approach to utilize an algorithm. More than one existing method might be apply to a particular problem, many a time an algorithm developed by one approach provide better solution than one, constructed using alternative techniques.

Few existing techniques areas below:

Brute Force

Brute force is an approach to directly solve a problem based on the given problem's statement, set of inputs, expected output and definitions of the concepts involved. Mostly they are useful for small domains, or we may say for simpler problems, due to overheads in sophisticated approaches.

It is sometimes less efficient than other techniques in the general case.

Some examples of brute force algorithms are:

Computing an (a > 0, n a non-negative integer) by multiplying a*a*…*a

Computing n!

Selection sort

Bubble sort

Sequential search

Exhaustive search: Traveling Salesman Problem, Knapsack problem.

Greedy Algorithms

Greedy Algorithms “take what you can get now” strategy

A greedy algorithm repeatedly executes a procedure which tries to maximize the return based on examining local conditions, with the hope that the outcome will lead to a desired outcome for the global problem. Now and again such a strategy is ensured to offer optimal solutions in a localized manner, and in some different cases, it might provide a compromise that produces acceptable approximations.

Typically, the greedy algorithms employ the strategies that are simpler to implement and require the minimal amount of resources.

Examples:

Minimal spanning tree

Shortest distance in graphs

Greedy Algorithm for the Knapsack problem

The coin exchange problem

Huffman trees for optimal encoding

Greedy techniques are mainly used to solve optimization problems. They do not always give the best solution.

It has been proven that greedy algorithms for the minimal spanning tree, the shortest paths, and Huffman codes always provide the optimal solution.

Divide-and-Conquer, Decrease-and-Conquer

These are methods of designing algorithms that took an instance of the problem to be solved, split this into several smaller sub-instances (of the same problem), independently solve each of the sub-instances and after that combine the sub- instance solutions to yield a solution for the original problem.

With the divide-and-conquer technique, the span of the problem instance is reduced by a factor (e.g. half the input size), while with the decrease-and-conquer method the size is reduced by a constant.

Examples of divide-and-conquer algorithms:

Computing an (a > 0, n a nonnegative integer) by recursion

Binary search in a sorted array (recursion)

Merge sort algorithm, Quick sort algorithm (recursion)

The algorithm for solving the fake coin problem (recursion)

Examples of decrease-and-conquer algorithms:

Insertion sort

Topological sorting

Binary Tree traversals: in order, preorder and post order (recursion)

Computing the length of the longest path in a binary tree (recursion)

Computing Fibonacci numbers (recursion)

Reversing a queue (recursion)

Warshall's algorithm (recursion)

Dynamic Programming

Dynamic programming is a favor name for using the divide-and-conquer technique with a table. It is a stage-wise search method suitable for optimization problems whose solutions might be seen as the consequence of a succession of choices.

The basic idea of dynamic programming is: avoid calculating the same stuff twice, for the most part by keeping up a table of known results of sub problems.

The dynamic programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of caching sub-problem solutions and appealing to the “principle of optimality.”

Dynamic Programming is a Bottom-Up Technique in which the smallest sub- instances are explicitly solved first and the consequences of these are used to develop solutions to progressively larger sub-instances.

In contrast, Divide-and-Conquer are a Top-Down Technique which logically advances from the initial problem split down to the smallest sub-instance via intermediate sub-instances.

Examples:

Fibonacci numbers computed by iteration.

Warshall's algorithm implemented by iterations

Transform-and-Conquer

Transform and Conquer is an algorithm design technique which works in two stages;

STAGE 1: (Transformation stage): The problem's instance is modified, that is more amenable to the solution.

STAGE 2: (Conquering stage): In this step the transformed problem is solved.

The three main variations of the transform & conquer differ by the way a given instance is transformed:

Instance Simplification: Transformation of an instance of a problem to an instance of the same problem with some specific property that makes the problem easier to solve.

Example:

list pre-sorting

AVL trees

Representation Change: Changing one representation of a problem's instance into another representation of the same instance.

Example:

2-3 trees,

Heaps and heap sort

Problem Reduction: Transforming a problem given to another one, that can be solved by a known algorithm.

Example:

Reduction to graph problems

Reduction to linear programming

Backtracking

For many real-world problems, the solution process comprises of working your way through a sequence of decision points in which every choice drives you further along some path. On the off chance that you make the correct set of decisions, you end up with the solution. On the other hand, if you reach a dead end or otherwise find that you have made an incorrect choice incidentally, you need to backtrack to a past decision point and attempt analternate path that may lead you to the solution. Algorithms that are implemented using this approach are called backtracking algorithms .

Backtracking uses depth-first search usually without cost function.

Example:

Solving Puzzles such as eight queens puzzle, crosswords, verbal arithmetic, Sudoku, Peg Solitaire.

Combinatorial optimization problems such as parsing and the knapsack problem.

Logic programming languages such as Icon, Planner and Prolog, which use backtracking internally to generate answers.

Branch-and-bound

Generally, Branch and bound is used when we represent our problem in a state- space tree form in a breadth-first manner.

Then evaluate each node of this tree using the cost and utility functions. By evaluating node at each step, we choose the best node to proceed further. Branch- and-bound algorithms are implemented by using a priority queue.

Example: The 8 queen problem and puzzle problem. The cost function is the number of moves. The utility function evaluates how close a given state of the puzzle to the goal state is, e.g. counting how many queens are not in place.

Specifications

There are various ways by which we can specify an algorithm-

All the ways have discussed in the above topics.

Algorithmic Verifications

Algorithmic verification means checking the validity of algorithm. We can check it for a set of correct outputs by giving a valid set of input values for a finite period of time. A proof of algorithm can be complex at many a times. To show that algorithm is not working properly we have to show that for at least a given input instance it is giving an incorrect value.

Analysis of algorithm

To choose the best algorithm for a particular task, we need to be able to judge how long a particular solution will take to run. To analyze an algorithm is to determine the amount of resources (such as time and storage) necessary to execute it. Usually, the efficiency or complexity of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space or memory complexity) required to execute the algorithm.

Implementation of algorithm

Implementation of the algorithm should be done in appropriate language. We should write an optimal code to reduce the burden of the compiler.

REVIEW EXERCISE

Define algorithm. Explain the role of the algorithm in various fields.

State and explain the properties of an algorithm.

What are the various stages needed to be followed while designing and analysis of an algorithm?

What is the analysis of an algorithm?

What is the complexity of an algorithm? Explain time and space complexity?

Write an algorithm to find whether a given number is prime or not.

Write an algorithm to calculate Fibonacci series up to the nth number.

Write an algorithm to find the greatest number in an array.

SOLUTION OF REVIEW EXERCISE

Ans. 6. Program Prime

i. Read A

ii. Compute B ← A- 1

iii. IsPrime ← True

iv. While (B != 1)

  a. If (A/B gives no reminder) Then
     I IsPrime ← False

  b. EndIf

  c. Compute B ← B – 1

v. EndWhile

vi. If (IsPrime = = True) Then

  a. Display "Number is prime"

vii. Else

  a. Display "Number is not prime"

viii. EndIf

ix. End.

Output

1. Read 5
2. B ← 5 – 1 = 4
3. IsPrime ← True
4. While (4! = 1)
   a. If (5 / 4 gives no reminder.)
   c. B ← 4-1=3

Again go to step 4. While (3 ! = 1)

   a. If (5 / 3 give no reminder)
   c. B ← 3- 1= 2

Again go to step 4. While (2 ! = 1)

   a. If (5 / 2 give no reminder)
   c. B ← 2- 1= 1

Again go to step 4. While (1 ! = 1)

5. EndWhile
6. If (IsPrime = = True) Then
   a. Display "Number is prime"
8. EndIf
9. End

Ans 7. Program Fibonacci

1. Read n
2. x ← 0
3. y ← 1
4. For i← 2to n do
   a. z←x + y
   b. x←y
   c. y←z
   d. Display value of z
   e. i←i + 1
5. EndFor
6. End

Output

4. For i←2
   a. z←0 + 1 = 1
   b. x←1
   c. y←1
   d. Display 1
   e. i←2 +1= 3

Again go to step 4. For i←3
   a. z ←1 + 1 = 2
   b. x ←1
   c. y←2
   d. Display 2
   e. i←3 + 1 = 4

Again go to step 4. For i←4
   a. z←1 + 2 = 3
   b. x←2
   c. y←3
   d. Display 2
   e. i←3 + 1 = 4

Again go to step 4. For i←4
   a. z←2 + 3 = 5
   b. x←3
   c. y←5
   d. Display 5
   e. i←4 + 1 = 5

Ans 8. Program Greatest Element In Array

1. Read size
2. For i←0 to size-1 do
   a. Read a[i]
   b. i←i + 1
3. EndFor
4. large←a[0]
5. For i←1 to size -1 do
   a. If (large< a[i]) Then
   b. large←a[i]
   c. End If
   d. i←i + 1
5. EndFor
6. Display large
7. End

Output

1. Read 4
2. For i←0 to 3 do
   a. Read a[0]←2, a[1]←4, a[2]←6, a[3]←3
4. large←2
5. For i←1
   a. If (2 < 4) Then
   b. large←4
   d. i←1 + 1 = 2

Again go to step 5. For i←2
   a. If (4 < 6) Then
   b. large←6
   d. i←2 + 1 = 3

Again go to step 5. For i←3
   a. If (6 < 3) Then
   d. i←3 + 1 = 4

6. Display 6

C HAPTER -2

Complexity of Algorithms

In this chapter student will understand:

What is the analysis of Algorithm? What is the role of analysis of algorithm? How one can compute the complexity of an algorithm?

2.1 Analysis of algorithm

More than one algorithm might work to perform the same operation, but some algorithms use more memory and take longer time to execute than others. Also, how would we realize that calculation will work better when all is done, given differences between computers and data inputs? This is how algorithm analysis comes into play.

One way to test an algorithm is to run a computer program and see how well it works. The problem with this approach is that it only tells us how well the algorithm does with a particular machine and set of inputs. The purpose of algorithm analysis is to check and then conclude how well a particular algorithm works in general. This analysis would be extremely troublesome and tedious undertaking to do on singular PCs, so specialists device models of PC is working to test algorithms.

In general, algorithm analysis is about discovering how much time a program takes to run (Running time), and how much memory storage it needs

to execute the same program (Memory space). Specifically, computer science researchers do algorithm analysis to decide how the information is input into a program which affects its aggregate running time, how much memory space needed for program information, how much space the program's code requires in the memory, regardless of whether an algorithm produces correct results, how complex a program is, and how well it manages the outcomes.

2.2 Complexity

The complexity of an algorithm is a function describing the efficiency of the algorithm regarding the amount of data the algorithm must process. There are measuring units for the domain and range of this function.

In simple words, the complexity of an algorithm is a measure of the amount of time and memory space required by an algorithm for an input of a given size (n).

By resources to be estimated, complexity is classified as below:

(1) Time Complexity

(2) Space Complexity

2.2.1 Time Complexity

Time complexity is a function describing the amount of time an algorithm takes regarding the number of inputs to the algorithm. “Time” measure the number of memory accesses performed, the number of comparisons between integers, the number of iterations in inner loop executes, since there are many factors unrelated to the algorithm that can affect the real time (like the language used, type of computing hardware, proficiency of the programmer, optimization in the compiler, etc.).

Syntax: T (A, n) where T is the number of elementary instructions executed, A is algorithm and n denotes the size of data input.

2.2.2 Space Complexity

Space complexity is a function describing the amount of memory (space) an algorithm takes regarding the number of inputs given to the algorithm. We often speak of “extra” memory required, never take into account the memory needed to store the input itself. Here, we use natural (but fixed-length) units to measure the space. We can use bytes, yet it is easier to utilize, the number of integers used, the number of fixed-sized structures, and so forth. At last, the function we think will be independent of the actual number of bytes expected to represent the unit. Space complexity is sometimes ignored because the space used is minimal but sometimes it becomes as important as time.

Syntax: S (n) where S is the space required by the set of functions computable in space, at most c*S (n) for some constant c>0, where n is input size.

2.2.3 Worst-case, Best-case, Average case complexities

Worst-case complexityz: complexity (number of times the basic operation executed) for the worst case is found for input of size n, hence the algorithm runs for longest time as compared to all possible inputs of size n.

If complexity is indicated by function f (n) then in worst case, it is indicated by the maximum value of f (n) for any possible input.

Best-case complexity: complexity (number of times the basic operation executed) for the best case is calculated for input of size n., when the algorithm runs the fastest as compared with all possible inputs of size n.

Here, the value of f(n) is minimum for any possible input.

Average-case complexity: Average time was taken (number of times the basic operation executed) to solve all the possible instances (random) of the input.

The value of f (n) lies in between maximum and minimum for any possible input.

Note: Here the average doesn't mean by the average of worst and best case.

2.3 The Asymptotic Notations

We are usually interested in the order of growth of the running time of an algorithm, not in the exact running time. These notations are also used to represent the asymptotic running time.

We have to build up an approach to discuss the rate of growth of functions so that we can compare algorithms. Asymptotic notation gives us a method for classifying the functions according to their rate of growth.

The Asymptotic notation is a shorthand way to write down and talk about 'the fastest possible' and 'the slowest possible' running times for an algorithm, utilizing highest and lowest bounds on speed.

It mainly measures computation time of any algorithm.

The primary Asymptotic Notations are:

θ (Theta) Notation [Average number of steps to solve a problem, (used to express both upper and lower bound of a given f (n)]

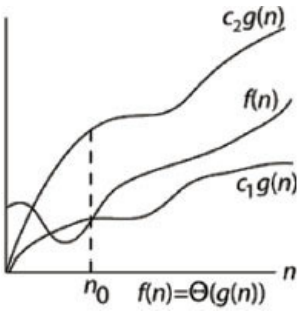
O (Big-"Oh") Notation [Maximum number of steps to find a solution to the problem, (upper bound)]

Ω (Big-"Omega") Notation [Minimum number of steps to execute the problem, (lower bound)]

2.3.1 Theta Notation (θ-Notation)

If, $0 \le C_1.g(n) \le f(n) \le C_2.g(n)$ then $f(n) = \theta(g(n))$ for all $n \ge n_0$.

Here f (n) and g (n) are two functions, n is the set of positive integers, and n 0 is an input till which the above-given condition may or may not follow. But after it, the relation is always valid. As shown in the graph below:



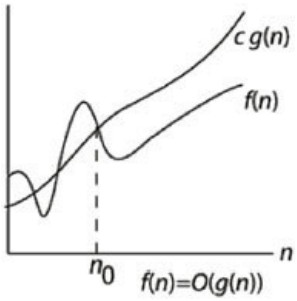
θ- Notation is "asymptotically tight bound" because of its equivalence from both upper and lower bound.

2.3.2 Big-Oh Notation (O-Notation)

If, $0 \le f(n) \le C_2.g(n)$ then $f(n) = O(g(n))$ for all $n \ge n_0$.

(Read it as "f of n is big oh of g of n" or "f is big oh of g")

Here f (n) and g (n) are two functions, n is the set of positive integers, and n 0 is an input value till which the above given condition may or may not follow. But after it, the relation is always valid, as shown in the graph below.

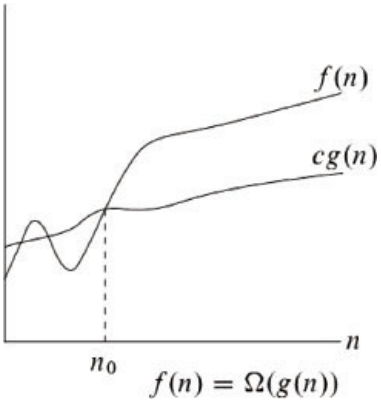


O-notation is a "tight upper bound notation", and it provides maximum value of running time for any given function.

2.3.3 Omega-Notation (O-Notation)

If, $0 \le C_1.g(n) \le f(n)$ then $f(n) = \Omega(g(n))$ for $n \ge n_0$.

Here f (n) and g (n) are two functions, here n is the set of positive integers, and n 0 is an input till which the above-given condition may or may not follow. But after it, the relation is always valid, as shown in the graph below.



$f(n) = \Omega(g(n))$

Ω-notation is a “tight lower bound notation”, and it provides the non-negative minimum value of running time for any given function.

Note: If θ g (n) exist for any function then both O g(n) and Ω g(n) exist but if either O g(n) or Ω g(n) exists then it is not necessary that q g (n) will exist.

2.3.4 Little oh-Notation (o-Notation)

If, $0 \leq f(n) < C_2 . g(n)$ then $f(n) = o(g(n))$ for all $n \geq n_0$.

f (n) becomes insignificant related to g (n) as n approaches to infinity:

$$o(g(n)) : \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

We use o-notation to denote a loose upper bound that is “NOT asymptotically tight” because of no equivalence in its relation with f(n).

2.3.5 Little Omega Notation (ω-Notation)

If, $0 \leq C_1 . g(n) < f(n)$ then $f(n) = \theta(g(n))$ for $n \geq n_0$.

When f(n) becomes arbitrarily large relative to g(n) as n approaches infinity:

$$(\omega)(g(n)) : \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

We use θ-notation to denote weaker (non-asymptotically tight bound) lower bound because of no equivalence in its relation with f(n).

2.4 Relational Properties to be applied on Asymptotic Notations:

Here we assume that all the functions are asymptotically positive.

Transitivity

[A relation R is transitive iff x is related by relation R to y and y is related by the same relation R to z then x is also related by R to z.]

a) If, $f(n) = \theta(g(n))$ && $g(n) = \theta(h(n))$ => $f(n) = \theta(h(n))$
b) If, $f(n) = O(g(n))$ && $g(n) = O(h(n))$ => $f(n) = O(h(n))$
c) If, $f(n) = \Omega(g(n))$ && $g(n) = \Omega(h(n))$ => $f(n) = \Omega(h(n))$
d) If, $f(n) = o(g(n))$ && $g(n) = o(h(n))$ => $f(n) = o(h(n))$
e) If, $f(n) = \omega(g(n))$ && $g(n) = \omega(h(n))$ => $f(n) = \omega(h(n))$

Note: Transitivity is applicable to all notations (θ, O, Ω, o, ω).

Reflexivity

[A relation R is reflexive iff everything bears R to itself, i.e., if x is related by R to x then x is related by R to x.]

(a) $f(n) = \theta(f(n))$

(b) $f(n) = O(f(n))$

(c) f(n) = Ω(f(n))

Note: Reflexivity is applicable to main notations only (θ, O, Ω).

Symmetry

[A relation R is symmetric iff x is related by relation R to y, then y is related by the same relation R to x.]

f(n) = θ(g(n)) iff g(n) = θ(f(n))

Note: Symmetricity is applicable only to -notation as reverse equality exist only in this relation.

Transpose Symmetry

[A relation R is transpose symmetric iff x is related by R to y, then y is related by R to inverse of x.]

(a) f(n) = O(g(n)) iff g(n) = Ω(f(n))

(b) f(n) = o(g(n)) iff g(n) = ω(f(n))

Note: Transpose Symmetry is applicable to extreme reverse notations, i.e., O and o.

2.4.1 Examples based on properties

Example1: Prove that $f(n)+of(n)=\Theta f(n)$

is true.

$$f(n)+of(n)=\Theta f(n)$$

As from definition of little “oh” we have

$$\begin{aligned} & g(n)=of(n) \\ & 0 \le g(n)<c.f(n) \\ \Rightarrow \quad & f(n) \le f(n)+g(n)<(1+c)\, f(n) \\ \Rightarrow \quad & f(n)+g(n)=\Theta f(n) \\ \Rightarrow \quad & f(n)+of(n)=\Theta f(n) \end{aligned}$$

Hence, it is true.

Example 2: Prove that 2: = ω(2 n)

As from definition of little “omega” we have

$$\begin{aligned} & \Rightarrow c.\, g(n)<f(n) \\ & \Rightarrow c.2^n < 2^{2n} \\ & \Rightarrow c.2^n < (2^n)^2 \\ & \Rightarrow c. < 2^n \end{aligned}$$

Which is true as 2 n is an exponent for every value of n, it is always greater than a constant.

2.5 Standard notations and common functions:

Monotonicity: A function f (n) is monotonically increasing, if for $m \le n \Rightarrow f(m) \le f(n)$

and function f (n) is monotonically decreasing, if for. $m<n \Rightarrow f(m) \ge f(n)$

Similarly, we say that A function f (n) is strictly increasing, if for $m<n \Rightarrow f(m) < f(n)$

and function f (n) is strictly decreasing, if for $m<n \Rightarrow f(m) > f(n).$

Floor and ceiling: For any real integer x, we denote greatest integer less than equal to x by floor, denoted by $\lceil x \rceil$

similarly for any real integer x, we represent least integer greater than equal to x by the ceiling, as $\lfloor x \rfloor$

these functions are monotonically increasing.

We may further represent it by the equation -

$x\text{-}1 \leq \lceil x \rceil \leq x \leq \lfloor x \rfloor \leq x+1$

For any integer n, we have - $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$

For any real number n ≥0 and integers a> 0 and b >0

- $\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil$
- $\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor$

Modular arithmetic : For any integer a and any positive integer n the value of a and n Remainder of (a/n)

if a mod = b mod n ⇒a = b mod or we may say b is equivalent to a.

Polynomial: For a positive integer d , a polynomial p (n) of n of degree d is given by

$p(n) = \sum_{i=0}^{d} a_i n^i$

Where a i is the non-zero negative integer. A polynomial is asymptotically positive if and only if a d > 0. For an asymptotically positive polynomial p(n) of degree d, we have $p(n)=\Theta(n^d)$

.

Exponential: For all real a > 0, m , and n , we have the following identities:

a º = 1

a ¹ = a

a -1 = 1/ a

(a m) n = (a nm)

(a m) n = (a n) m

a m a n = a m+n

For all n and a ≥1, the function a n is monotonically increasing in n . When convenient, we shall assume 0 º = 1.

Logarithmic: For logarithmic function, we may use following notations-

For all real numbers a,b,c > 0

$$a = b^{\log_b}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b 1/a = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

In each formula logarithmic base is not equal to 1.

Factorial: Factorial of an integer n , where n≥0 is denoted by n !.

A weak upper bound on the factorial function is n ! ≤ n n , since each term in the factorial is at most n .

Stirling's approximation ,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Where e is the base of the natural logarithm, gives us a tighter upper bound as well as a lower bound.

We can also define

$$n! = o(n^n)$$

$$n! = \omega(2^n)$$

$$\lg(n!) = \Theta(n \lg n)$$

2.6 Basic Efficiency Classes

Fast

1	Constant
Log n	Logarithmic
N	Linear
N Log n	N log n
N^2	Quadratic
N^3	Cubic
N^k	Polynomial
2^n	Exponential
N!	factorial

Slow

High Time Efficiency

Low Time Efficiency

Seeing the above classes, we can guess about different results for same input values. For example, let n=2, 6, 150….etc. Now, the growth rate of various functions has shown below:

1	1	1	1
Log n	0.301	0.778	2.176
N	2	6	150
N Log n	0.602	4.669	326.41
N^2	4	36	22500
N^3	8	216	3375000
N^k(if k=5)	32	7776	75937500000
2^n	4	64	10^{45}
N!	2	720	$>10^{50}$

In the above given table we can see that for the higher value of n, the resulting value of each function follows the order in which they have arranged. However, for small values, the order does not retain its values. We can conclude one more thing by this result that, “Algorithms are always analyzed for large input values not for the smaller oness”.

Hierarchy of functions:

$0 < 1/n < 1 < \log\log\log n < \log\log n < \sqrt{\log n} < \log n < \log^2 n < \log^3 n < \sqrt{n} < n < n\log n < n^2 < n^2\log n < n^3 < 2^n < n.2^n < 3^n < n! < n^n < (2^2)$ (2 power n)

2.7 Logarithmic Rules

1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \log_b a$
4. $\log_b a = (\log_c a)/\log_c b$
5. $b^{\log_c a} = a^{\log_c b}$
6. $(b^a)^c = b^{ac}$
7. $b^a b^c = b^{a+c}$
8. $b^a/b^c = b^{a-c}$

2.7.1 Examples based on rules

1. $\log(2n \log n) = 1 + \log n + \log\log n$ (rule 1)
2. $\log(n/2) = \log n - \log 2 = \log n - 1$ (rule 2)
3. $\log \sqrt{n} = \log(n)^{1/2} = (\log)/2$ (rule 3)
4. $\log\log \sqrt{n} = \log(\log n)/2 = \log\log n - 1$ (rule 2, rule 3)
5. $\log_4 n = (\log n)/2$ (rule 4)
6. $\log 2^n = n$ (rule 3)
7. $2^{\log n} = n$ (rule 5)
8. $2^{2\log n} = (2^{\log n})^2 = n^2$ (rule 5, rule 6)
9. $4^n = (2^2)^n = 2^{2n}$ (rule 6)
10. $N^2\, 2^{3\log n} = n^2 . n^3 = n^5$ (rule 5, rule 6, rule 7)
11. $4^n/2^n = 2^{2n}/2^n = 2^{2n-n} = 2^n$ (rule 6, rule 8)

2.8 Recurrence Relation:

A recursive or inductive definition has two parts, discuss as follows:

Base Case : The initial condition or step which defines the very first (or first few) element of the sequence.

Inductive (Recursive) Case : An inductive step is one in which later values are in the sequence and defined in terms of earlier calculated values of the sequence.

A recurrence method is a function defined in term of some base cases. The algorithms having recursion are solved by recurrence relation to calculate their running time complexity.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \qquad \text{where a} \geq 1 \& \text{ b}>1$$

Here T(n) is a problem definition,

where a≥1& b>1

(n/b) represent the number of elements and

f (n) represents cost.

2.8.1 Methods to Solve Complexity of Recurrence Relations

Iteration Method

Substitution Method

Recursion-tree Method

Master Method

2.8.1.1 Iteration Method- In the iteration method we iteratively “unfold” the recurrence until we “see the pattern.”

The iteration method does not require making a good guess like the substitution method (but it is often more involved than using induction)

Example: Solve recurrence T (n)= T (n +1)+ n where $T(n)=\begin{cases}1 \text{ if } n=1\\ T(n-1)+n\end{cases}$

using iteration method.

$T(n) = T(n\text{-}1)+n$ **eq(1)**

Replacing value of $T(n\text{-}1)$ in eq (1) $\left\{\begin{array}{l}\text{finding value of } T(n-1) \text{ from } Eq(1)\\ \Rightarrow T(n-1)=T(n-2)+(n-1)\end{array}\right\}$

$T(n)=T(n\text{-}2)+(n\text{-}1)+n$ **eq(2)**

Replacing value of $T(n\text{-}2)$ in eq(2) $\left\{\begin{array}{l}\text{Adding finding value of } T(n-2) \text{ from } eq(1)\\ T(n-2)=T(n-3)+(n-2)\end{array}\right\}$

$T(n) = T(n\text{-}3)+(n\text{-}2)+(n\text{-}1)+n$

Similarly, if we simplify it for all values, we get

$T(n) = T(n\text{-}k)+(n\text{-}(k\text{-}1))+(n\text{-}(k\text{-}2))+\ldots\ldots\ldots\ldots+n$

Let $n\text{-}k=1$

$\Rightarrow T(n)=T(1)+2+3+\ldots\ldots\ldots\ldots+n$

$\Rightarrow T(n)=(1)+2+3+\ldots\ldots\ldots+n$

$$=\frac{n(n+1)}{2} \cong O(n^2)$$

Example: Solve recurrence T (n)= T (n +2)+ c where $\begin{cases}1 \text{ if } n=1\\ T(n-1)+n\end{cases}$

using iteration method.

$$T(n) = T(n/2) + c \qquad \textbf{eq(1)}$$

Replacing value of $T(n/2)$ in eq (1) $\left\{ \text{finding vlaue of } T\left(\frac{n}{2}\right) \text{ from } Eq(1) \Rightarrow T\left(\frac{n}{2}\right) = \left(\frac{n}{4}\right) + c \right\}$

$$T(n) = T\left(\frac{n}{4}\right) + c + c \qquad \textbf{eq(2)}$$

Replacing value of $T(n/4)$ in eq(2) $\left\{ \text{again finding vlaue of } T\left(\frac{n}{4}\right) \text{ from eq}(1) \Rightarrow T\left(\frac{n}{4}\right) = T\left(\frac{n}{2^3}\right) + c \right\}$

$$T(n) = T\left(\frac{n}{2^3}\right) + c + c + c$$

Similarly if we simplify it for all values, we get

$$T(n) = T\left(\frac{n}{2^k}\right) + kc$$

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

$$n = 2^k$$

$$\log n = k.\log 2$$

$$\Rightarrow k = \log n$$

$$T(n) = T(1) + kc$$

$$T(n) = 1 + kc = 1 + c.\log n$$

$$\Rightarrow T(n) \simeq \mathbf{O}(\log n)$$

2.8.1.2 Substitution Method

This method is “guess and work” method!

The substitution method consists of following steps:

(a) Guess the form of the solution.

(b) Perform mathematical induction to prove the guess.

This method is used mainly to find upper or lower bound of a recurrence relation.

Note: If the guess proves wrong, adjust the guess and apply again.

Example: Find the solution of the recurrence relation T (n)= T (n -2)+1, which is the result of an algorithm meant to find the summation of odd integers.

As mentioned that the recurrence relation is only for odd integers, therefore n denotes odd integers, i.e., n =1,3,5…. n

Given T(n-2)+1

$$\text{If } n=1 \text{ then}$$

$$T(1)=T(1-2)+1$$

$$\Rightarrow T(1)=1$$

Step 1: Guess

$T(n)=O(n)$

$\Rightarrow T(n) \leq c.n$

Step 2: if $n=3$

$T(n) \leq c.n$

$T(n-2)+1 \leq c.n$

$T(3-2)+1 \leq c.3$

$1+1 \leq 3c$

$c \geq 2/3$

& $n_0 = 3$

Example: Solve the recurrence relation T (n)=2 T (n /2)+ n , using substitution method.

Step 1: Guess

$T(n)=O(n \log n)$

$\Rightarrow T(n) \leq c.n \log n$

Where $c > 0, n \geq n_0$

Step 2: induction

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

$$T(n) = 2c.\frac{n}{2}\log n/2 + n$$

$$T(n) = c.n.\log n - c.n.\log 2 + n$$

$$T(n) = c.n.\log n - (c-1)n$$

=Desired – Residue

$\Rightarrow T(n) \leq c.n.\log n$

$(c-1).n \geq 0$

Where c ≥1 &≥1

2.8.1.3 Recursion Tree

In recursion trees each node represents the cost of a single sub problem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level cost, and then we sum all costs, per level, to determine the total cost for all levels of recursion. No doubt iteration is a powerful tool for solving recurrences. Many a times it loses sight of what is going on. Any recurrence can be represented in the form of tree, where each expansion takes us to one level deeper in the tree. To visualize the actual working of iteration we use recursion tree method.

Example: $T(n)\begin{cases}1 & if\ n=1\\ 2T\left(\frac{n}{2}\right)n+n\ else\end{cases}$

Run time tree

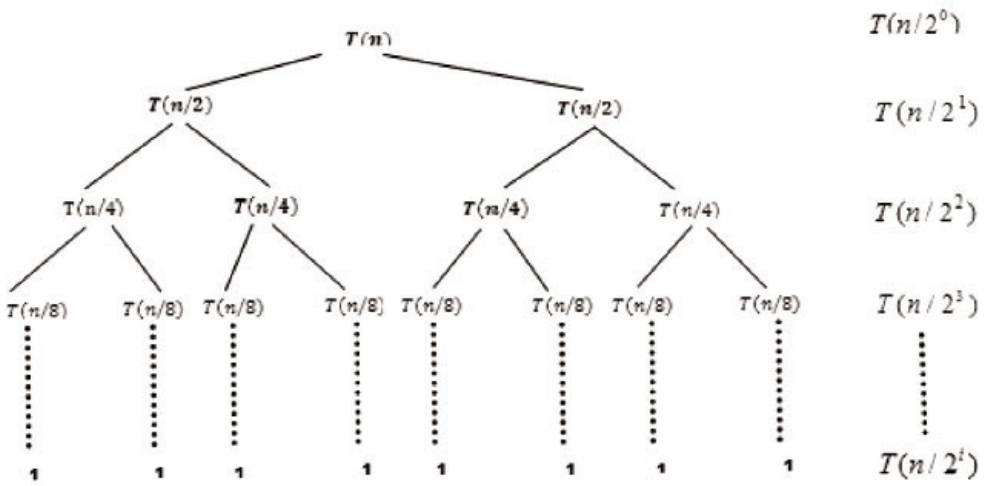
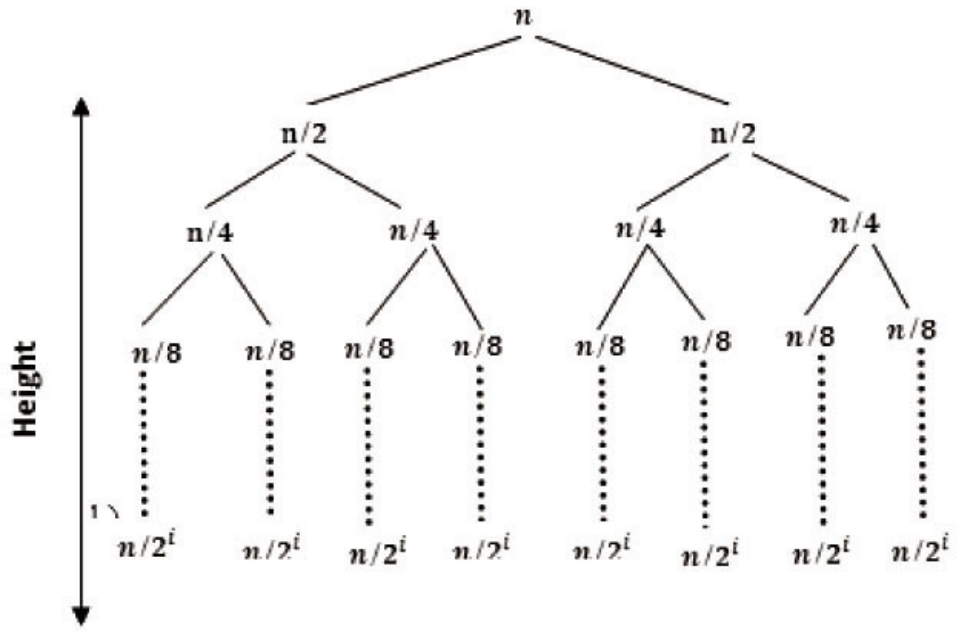


Fig.: Run time



$\Rightarrow T(n/2^i)=T(1)$

$\Rightarrow n/2^i)=1$

$n=2^i$

Or $i = \log_2 n$ where i represent height

Cos $t = i.n=n.\log_2 n$

cos $t = 0(n \log n)$

Hint: For calculating height we can use Arithmetic progression (A.P.), geometric progression (G.P.) and Harmonic progression (H.P.), formulas for these progressions are as follows

Arithmetic progression (A.P.): Series will be in form of a, a+d, a+2d….

$$a_n = a+(n-1)d$$

Geometric progression (G.P.): Series will be in form of a, ar, ar^2, ar^3…

$$a_n = ar^{(n-1)}$$

Harmonic progression (H.P.): Series will be reciprocal to A.P. and can be solved by converting it into A.P.

2.8.1.4 Master Method

We usually refer to master method as a cookbook of recurrence relations where we can get the solution based on the predefined issues. The Master method is derived from recursion tree method. In recursion tree, we calculate the work done at each level and sometimes the resulting output is in polynomial form. However, in the master method we get the solution directly. Master method applies on the equation in the form of

$$T(n) = \alpha T\left(n/b\right) + f(n) \text{ where } a \geq 1, b > 1$$

The master method requires memorization of three cases, but then the solution of many recurrences can be determined quite easily, often without pencil and paper.

There are following three cases

1. If $f(n) = O(n^{\log_b a - \epsilon})$ where $\epsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ where $\epsilon > 0 \Rightarrow T(n) = \Theta(f(n))$ & $af(n/b) \leq cf(n)$

Example: Solve the recurrence T (n) = 3 T (n /4)+ n log n where a=3, b=4 and f (n)= n log n

$$\Rightarrow n^{\log_b a} = n^{\log_4 3}$$

$$n \log n \geq n^{\log_4 3} \Rightarrow f(n) = \Omega(n^{\log_4 3})$$

$$af(n/b) \leq cf(n)$$

$$3f(n/4) \leq cn \log n$$

$$3.\frac{n}{4}\log(n/4) \leq cn \log n$$

As III[rd] condition $\frac{3}{4}(n \log n - n \log 4) \leq cn \log n$

$$\frac{3}{4}n(\log n - 2) \leq cn \log n$$

$$\frac{3}{4}n \log n - \frac{3}{2}n \leq cn \log n$$

That is also true.

$$\therefore T(n) = \Theta(\log n)$$

2.9 Sorting Techniques

2.9.1 Insertion Sort

Insertion sort is used by everyone in his real life more than any other algorithm. If you have ever reorganized your book shelf, the arrangement of playing cards, etc. These types of activities are the example of insertion sort in the real world. Let's discuss one more example. When we go for shopping in a shop, all the items are arranged in a pile according to their size like, XXL, XL, L, M, S, XS, etc. If we choose any of them according to our choice, we pull the item from the pile and after our trial, the salesperson again reorganizes it accordingly.

Pseudo code:-

INSERTION-SORT(A)	*cost*	*times*	
1 **for** $j \leftarrow 2$ **to** *length* [A]	$c1$	n	
2 **do** $key \leftarrow A[j]$		$c2$	n - 1
3 Insert $A[j]$ into the sorted sequence $A[1 \ldots j - 1]$.	0	n - 1	
4 $i \leftarrow j$ - 1	$c4$	n - 1	
5 **while** $i > 0$ and A[i] > key	c5	$\sum_{j=2}^{n} t_j$	
6 **do** $A[i+1] \leftarrow A[i]$	$c6$	$\sum_{j=2}^{n} (t_j - 1)$	
7 i ← i - 1	c7	$\sum_{j=2}^{n} (t_j - 1)$	
8 $A[i+1] \leftarrow key$	$c8$	$n-1$	

Here c1,c2,…. Are the cost for executing each statement and length [A]=n. Therefore statement run n times from 2 to n, statements from 2-4 all will run for n-1 times.

The total running time will be the sum of all running times (which will be their cost * time) i.e.

$$T(n) = c_1 * n + c_2(n-1) + c_4(n-1) + c_5 * \sum_{j=2}^{n} t_j + c_6 * \sum_{j=2}^{n} t_j - 1 + c_7 * \sum_{j=2}^{n} t_j - 1 + c_8(n-1)$$

$$T(n) = c_1 * n + (c_2 + c_4 + c_8)(n-1) + c_5 * \sum_{j=2}^{n} t_j + (c_6 + c_7) * \sum_{j=2}^{n} t_j - 1$$

Replacing values of $\sum_{j=2}^{n} t_j = \frac{n(n+1)}{2} - 1$ & $\sum_{j=2}^{n} (t_j - 1) = \frac{n(n-1)}{2} a$

We get equation $T(n) = c_1 * n + (c_2 + c_4 + c_8)(n-1) + c_5 * \frac{n(n+1)}{2} + (c_6 + c_7) * \frac{n(n-1)}{2}$

Which is in the form of T (n)= an ² + bn + c where a, b and c are constants.

Analysis of Insertion Sort: To insert the last element, i.e., n we need at most (n-1) comparisons and (n-1) iteration.

To insert the (n-1) th element i.e. we need at most (n-2) comparisons and (n-2) iteration. Similarly to insert 2nd element we need one comparison and one iteration.

Hence we have total iterations T(n) = 2*(1+2+3+…….+n-1)* = 2(n-1)*n/2= (n-1)*n

Best Case Analysis: In case of best case complexity, we will always have a sorted array. It means that we will always find, A[i] ≤ key.

⇒ T j =1

Insert the value of t j in the equation; we will get:

$$T(n) = c_1 * n + (c_2 + c_4 + c_8)(n-1) + c_5 * \sum_{j=2}^{n} 1 + (c_6 + c_7) * \sum_{j=2}^{n} 1 - 1$$

$$T(n) = c_1 * n + (c_2 + c_4 + c_8)(n-1) + c_5 * (n-1)$$

$$T(n) = O(n)$$

Hence its complexity is, T (n) = O(n).

Worst case Analysis: In the case of worst case complexity, we will always have an unsorted array or descending order array.

⇒ T j = j for j=2,3,….n.

Insert the value of t j in equation; we will get:

$$T(n) = c_1 * n + (c_2 + c_4 + c_8)(n-1) + c_5 * \sum_{j=2}^{n} j + (c_6 + c_7) * \sum_{j=2}^{n} j - 1$$

$$T(n) = c_1 * n + (c_2 + c_4 + c_8)(n-1) + c_5 * \left(\frac{n(n+1)}{2}\right) + (c_6 + c_7) * \left(\frac{n(n-1)}{2}\right)$$

$$\Rightarrow T(n) = \Theta(n^2)$$

Average Case Analysis: In the case of average case complexity, we will always have an unsorted array.

Hence average case complexity is, $T(n) = \Theta(n^2)$

2.9.2 Bubble sort

For better understanding the bubble sort we may take the example of schools, where teacher use to arrange students in the queue by order of increasing height. The way she did this was by standing in queue (in arbitrary order). Then she would start at the beginning of the line and compare the height of each student to that of the student behind him. If the student in front of him were taller than him, then she'd have the two students change places. She continued doing this until she reached the back of the line, taking note of whether or not two students had moved or not. If a swap had made, she would go back to the beginning of the line and start again. What teacher was doing (knowingly or unknowingly) was executing the Bubble Sort algorithm.

BUBBLE SORT(A)	Cost	Time
1 for i ← 1 to length[A]	c_1	n+1
2 do for j ← length[A] down to i + 1	c_2	$\sum_{j=2}^{n}(t_j - 1)$
3 do if A[j] < A[j - 1]	c_3	$\sum_{j=2}^{n}(t_j - 1)$
4 then exchange A[j] ← A[j - 1]	c_4	$\sum_{j=2}^{n}(t_j - 1)$

Now solve as we have previously solved; we got,

$$T(n) = c_1(n+1) + (c_2 + c_3 + c_4)\sum_{j=2}^{n}(t_j - 1)$$

$$T(n) = c_1(n+1) + (c_2 + c_3 + c_4)(\frac{n(n-1)}{2} - 1)$$

Analysis of Bubble Sort: Bubble sort is designed in such a manner that in the very first pass the largest number is placed and after this, in each pass through the list the next largest number moves to its right place. It implies that for each outer loop the inner loop will execute in as follows:

for 1 st iteration of the outer loop, the inner loop will execute = n times

for 2 nd iteration of the outer loop inner loop will execute- n-1 times.

It implies that the outer loop will execute total n time and for the n th iteration inner loop will execute

= n (n +1)

Hence the total execution will be T (n) = n * (1 + 2 + 3 + ……(n - 1) + n) = n * ∑ n = n (n +1)

Best Case Analysis: The best case for bubble sort occurs when we have already sorted list or nearly sorted one. In such case where the list is already sorted, bubble sort will terminate after the first iteration, since no swaps made. When a pass made through the list, and no swap takes place, it is certain that the list is completely sorted now.

Bubble sort is also efficient when one random value needs to be placed into a sorted list, provided that the new element is placed at the beginning and not at the end. When placed at the beginning, it will bubble up to the correct place, and the second iteration through the list, it will require no swap, which results in ending the iteration.

$$\Rightarrow t_j = 1$$

$$T(n) = c_1(n+1) + (c_2 + c_3 + c_4)\sum_{j=2}^{n}(1-1)$$

$$T(n) = c_1(n+1)$$

$$T(n) = O(n)$$

Worst Case: The absolute worst case situation for the bubble sort is when the smallest element of the list is at the large end. Because in each iteration, only the largest unsorted element gets placed into its proper location when the smallest value is positioned at the end, it is to be swapped each time through the list, and it won't reach the front of the list until all n iterations have occurred. In this worst case, it takes n iterations of n /2 swaps, so the order is, again, n ² .

$$\sum_{j=2}^{n} t_j - 1 = \frac{n(n-1)}{2}$$

On substituting this value in equation of T (n), we will get T (n)= O (n ²)

Average Case: This will be we same as the worst case complexity.

REVIEW EXERCISE

What do you understand by the complexity of an algorithm? Define time and space complexity of an algorithm?

How to find the complexity of algorithm using step count method? Explain any suitable example of iterative approach?

What is recurrence relation? Describe with the help of an example.

Solve recurrence T (n)= T (n)* n where $T(n) = \begin{cases} 1 if\ n = 1 \\ T(n-1) * n \end{cases}$

using iteration method.

Solve recurrence $T(n) = T(n\text{-}1) + \frac{1}{n}$

where $T(n) = \begin{cases} 1 \text{ if } n = 1 \\ T(n-1) * n \end{cases}$

using iteration

Solve the recurrence relation, T (n)= T (n -1)+ n , using substitution method.

Solve the recurrence relation T (n)=4 T (n /2)+ n , using substitution method.

Solve the recurrence relation $T(n)=3T(n/4)+\Theta n^2,$

using recursion tree method.

Solve the following recurrence relation

(i) T (n)= T (n -1)+5 for n >1, T (1)=0

(ii) T (n)=3 T (n -1) for n >1, T (1)=4

Write an algorithm to find time complexity of Fibonacci series with recursion tree method.

Write an algorithm to find time complexity of merge sort with recursion tree method.

Use the recurrence to solve T (n)= T (n /3)+ T (2 n /3)+ n .

Find complexity of $T(n)= 2T(\sqrt{n}) + \log n$

Find complexity of. $T(n)= T(\sqrt{n}) + 1$

Solve the recurrence relation $T(n) = \begin{cases} 2 & \text{for } n = 1 \\ 3T\left(\frac{n}{2}\right) + n \log_2 n & n > 1 \end{cases}$

Explain whether the relations given below are true or false-

(a) $f(n) = \Theta(f(n/2))$

(b) $o(g(n)) \cap \omega(g(n)) = \phi$

Use the master method to calculate tight asymptotic bounds for the following recurrences.

a. T(n) = 4T(n/2) + n.

b. T(n) = 4T(n/2) + n^2 .

T(n) = 4T(n/2) + n^3

Explain the solution for Tower of Hanoi problem?

MULTIPLE CHOICE QUESTIONS & ANSWERS

Which one of the following represents the tightest upper bound on the number of swaps required to sort n numbers using selection sort? (GATE - 2013)

(a) O(log n)

(b) O(n)

(c) O(n log n)

(d) O(n^2)

Ans. (b) O(n)

Which one of the following shows the tightest upper bound that represents the time complexity of inserting an object into a binary search tree of n

nodes? (GATE - 2013)

(a) O(1)

(b) O(log n)

(c) O(n)

(d) O(n log n)

Ans. (c) O(n)

The number of elements that can be sorted in Θ(log n) time using heap sort is (GATE - 2013)

(a) Θ(1)

(b) Θ(√log n)

(c) Θ(log n /(log log n))

(d) Θ(log n)

Ans. (c) Θ (log n /(log log n))

Consider the following function:

int unknown(int n){

int i, j, k=0;

for (i=n/2; i<=n; i++)

for (j=2; j<=n; j=j*2)

k = k + n/2;

return (k);} (GATE - 2013)

What will be the return value of the function is

(a) Θ(n ²)

(b) Θ(n ² log n)

(c) Θ(n ³)

(d) Θ(n ³ log n)

Ans. (b) Θ (n ² log n)

The recurrence relation for the optimal execution time of the Towers of Hanoi problem with n discs is (GATE-2012)

(a) T(n) = 2T(n - 2) + 2

(b) T(n) = 2T(n - 1) + n

(c) T(n) = 2T(n /2) + 1

(d) T(n) = 2T(n - 1) + 1

Ans.(d) T(n) = 2T(n - 1) + 1

Let W(n) and A(n) denote respectively, the worst case and average case running time of an algorithm executed on an input of size n. Which of the following statements is ALWAYS TRUE? (GATE 2012)

(a) A(n) = Ω (W(n))

(b) A(n) = Θ(W(n))

(c) A(n) = O(W(n))

(d) A(n) = o(W(n))

Ans.(c) A(n) = O (W(n))

Which of the given options provides the increasing order of asymptotic complexity of the given functions f1, f2, f3, and f4? (GATE-2011)

f 1(n)=2 n f 2(n)= n ³/² f 3(n)= n log2 n f 4(n)= n log2 n

(a) f 3, f 2, f 4, f 1

(b) f 3, f 2, f 1, f 4

(c) f 2, f 3, f 1, f 4

(d) f 2, f 3, f 4, f 1

Ans. (a) f 3, f 2, f 4, f 1

Two alternative packages A and B, having 10 k records are available for processing a database. Package A requires 0.0001 n ² time units, and package B requires 10 n log10 n time units to process n records. What is the smallest value of k for which package B will be preferred over A? (GATE-2010)

(a) 12

(b) 10

(c) 6

(d) 5

Ans. (c) 6

What is the number of swaps required, in the worst case to sort n elements using selection sort? (GATE-2009)

(a) Θ(n)

(b) Θ(n log n)

(b) Θ(n 2)

(d) Θ(n 2log n)

Ans. (a) Θ (n)

The running time of an algorithm is represented by the following recurrence relation:

T (n)= nT (n ³)+ cn n ≤3 otherwise

Which one of the following represents the time complexity of the algorithm? (GATE-2009)

(a) Θ(n)

(b) Θ(n log n)

(c) Θ(n ²)

(d) Θ(n ² log n)

Ans. (a) Θ (n)

Let w(n) and A(n) denote respectively, the worst case and average case running time of an algorithm executed on an input of size n. Which of the following is ALWAYS TRUE? (GATE CS -2012)

(a) A(n)=Ω(W(n))

(b) A(n)=θ(W(n))

(c) A(n)=O(W(n))

(d) A(n)=o(W(n))

Ans. (c) A(n)= O (W(n))

Explanation: The worst case time complexity is always greater than or same as the average case time complexity.

Consider the following functions

$f(n) = 3n^{\sqrt{n}}$

$g(n) = 2^{\sqrt{n \log_2}}$

$h(n) = n!$

Which of the following is true? (GATE CS 2000)

(a) h(n) = 0(f(n))

(b) h(n) = 0(g(n))

(c) g(n) ≠0(f(n))

(d) f(n) = 0(g(n))

Ans. (d) f(n) = 0(g(n))

Explanation: g(n) = 2^(√nlogn) = n^(√n)

f(n) and g(n) are of same asymptotic order and following statements are true.

f(n) = O(g(n))

g(n) = O(f(n)).

(a) and (b) are false because n! is of asymptotically higher order than n^(√n).

Consider the following pseudo code. Calculate the total number of multiplications to be performed?

D = 2

for i = 1 to n do

for j = i to n do

for k = j + 1 to n do

D = D * 3

(a) Half of the product of the three consecutive integers.

(b) One-third of the product of the three consecutive integers.

(c) One-sixth of the product of the three consecutive integers.

(d) None of the above. (GATE CS 2014)

Ans. (c) One-sixth of the product of the three consecutive integers.

The statement “D = D * 3” is executed n*(n+1)*(n-1)/6 times. Let us see how.

For i = 1, the multiplication statement is executed (n-1) + (n-2) + .. 2 + 1 times.

For i = 2, the statement is executed (n-2) + (n-3) + .. 2 + 1 times

……..

……..

For i = n-1, the statement is executed once.

For i = n, the statement is not executed at all

So overall the statement is executed following times

[(n-1) + (n-2) + .. 2 + 1] + [(n-2) + (n-3) + .. 2 + 1] + … + 1 + 0

The above series can be written as

S = [n*(n-1)*/2 + *(n-1)*(n-2)/2 + ….. + 1]

The sum of above series can be obtained by trick of subtraction the series from standard Series S1 = n2 + (n-1)2 + .. 12. The sum of this standard series is n*(n+1)*(2n+1)/6

S1 - 2S = n + (n-1) + … 1 = n*(n+1)/2

2S = n*(n+1)*(2n+1)/6 - n*(n+1)/2

S = n*(n+1)*(n-1)/6

Consider the following function

```
int unknown(int n) {
  int i, j, k = 0;
  for (i  = n/2; i <= n; i++)
    for (j = 2; j <= n; j = j * 2)
      k = k + n/2;
  return k;
}
```

What is the returned value of the above function? (GATE CS 2013)

(a) Θ(n2)

(b) Θ(n2logn)

(c) Θ(n3)

(d) Θ(n3logn)

Ans. (b) Θ (n2logn)

Explanation: The outer loop runs n/2 or Θ(n) times. The inner loop runs Θ(logn) times (Here j is divided by 2 in each and every iteration). So the statement “k = k + n/2;” runs Θ(nlogn) times. The statement increases value of k by n/2. So the value of k becomes n/2* Θ(nlogn) which is Θ(n2logn).

SOLUTION OF REVIEW EXERCISE

Ans 4. $O(n)!$ **Ans 5.** $O(\log 2n)$ **Ans 6.** $O(\log n)$ **Ans 7.** $O(n^3)$ **Ans 8** $\Theta(n^2)$

Ans 9.

Ans. **(i)** Let

T(n)=T(n-1)+5 =[T(n-2S)+5]+5

=T[(n-3)+5]+5+5 =T(n-3)+5*3

.....

= T(n-i)+5*i

.....

If i=n-1 then

= T(n-(n-1))+5*(n-1)

=T(1)+5*(n-1)

= 0 + 5(n-1) { T[1]=0 So T(n)=5(n-1)

Ans. **(ii)** T(n)=3 * T(n-1)

=3 * [3 * T(n-2)]

= 3*3 [3*T(n-3)]

=33 * T(n-3)

.....

=3i * T(n-i)

If we put i=n-1 then

=3(n-1) * T(n-(n-1))

= 3(n-1) * T(1)

T(n)= 3(n-1) * 4 { T(1)=4

Ans 10.

$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-1} + F_{i=2}$

for $i \geq 2$

Ans 11. For simplicity, we will assume that n is a power of 2. Each divide step yields two sub problems, both of size exactly n/2. The base case occurs when n = 1. When n≥2, time for merge sort steps are given below:

Divide: Just compute q as the average of p and r. D(n) = O(1).

Conquer: Recursively solve two sub-problems, each of size n/2. Therefore, total time is 2T (n/2).

Combine: MERGE on an n-element sub array takes O(n) time. Therefore, C(n) = O(n). Since D(n) = O(1) and C(n) = O(n), summed together they give a function that is linear in n: O(n). Recurrence for merge sort running time is

T(n) = O(1) if n = 1,

T(n) = 2T (n/2) + O(n) if n≥2.

T(n) = O(n log n).

Ans 12. As here we find that T (n)= lst term+2.lst term + n hence there will be two tree one having child node of double weight age of another tree. So here we consider only bigger tree and calculate cost according to it. i.e. cost=Θ(nlog 2 n)

Ans 13. $T(n) = 2T(n^{1/2}) + \log n$

Let $n=2^m$

$\log n = m \log 2$

$\Rightarrow n = 2^m$

$n^{1/2} = 2^{m/2}$

$\therefore T(2^m) = 2T(2^{m/2}) + \log 2^m$

$T(2^m) = 2T(2^{m/2}) + m \log 2$

$T(2^m) = 2T(2^{m/2}) + m$

Let $T(2^m)=\delta(m)$

$\Rightarrow T(2^{m/2}) = \delta(m/2)$

$\delta(m) = 2\delta(m/2) + m$

$Complexity = \Theta(m \log m)$

$\Rightarrow Complexity = \Theta(\log n \log \log n)$

Ans 14. Cost = $\theta(log_2 m)$ Complexity= $\Theta(log(logn))$

Ans 15. $T(n) = \theta n^{\log_2 3} - 6n - 2\log_2 n = O(nh^2)$

Ans 16. a) False **b)** true

C HAPTER -3

Divide-and-Conquer Algorithms

In this chapter student will understand:

What is divide and conquer strategy? In which cases it is better than brute force method? What are the application areas of divide and conquer?

3.1 Introduction

Divide and conquer is a general algorithm design strategy. It is a top-down technique for outlining algorithm. Its general idea is to break a problem into smaller sub - problem, finding the solution of those sub - problems and then combining all the solutions of sub - problems into the final result.

Formally, steps of Divide and conquer algorithm:

Divide: Break the given problem into smaller problems of the same type.

Conquer: Recursively take care of these smaller problems where the base case for the recursion is sub-problem of consistent size.

Combine: Appropriately combine the answers

3.2 General Divide & Conquer Recurrence

An instance of size ‘n’ can be broken down into ‘b’ instances of size ‘n/b,’ with “a” times of them needed to be solved. [a≥1, b > 1]. Assume size n of the problem is a multiplier of b.

The recurrence for the running time T (n) is as follows:

T (n) = a*T (n/b) + f(n)

Where: T (n/b) is time for each sub-problem;

f(n) - a function that accounts for the time spent on dividing the problem into smaller ones and on joining their answers.

Therefore, the order of growth of T (n) relies upon the values of the constants a & b and the order of growth of the function f(n).

3.3 Median and Order Statistics

The i th order statistic of a set of n elements is the i th smallest element. For instance, the minimum of a set of elements is the first order statistic (i = 1), and the greatest is the n th order statistic (i = n). A median, casually, is the “midpoint” of the set.

At the moment, when number of elements in a set are odd, the median is unique, occurring at i = (n + 1)/2. Similarly when number of elements in

the set is even then there may be two medians, happening at i = n/2 and i = n/2 + 1. Therefore, regardless of the parity of n, median occur at position $i = \lfloor (n+1)/2 \rfloor$

and $i = \lceil (n+1)/2 \rceil$.

Example: For a given set A of n (distinct) numbers and a number i , given such that 1≤ i ≤ n . We find a number x which is larger than exactly i-1 elements of a set.

Analysis: This selection can be made in O(n log n) time.

3.4 Binary Search

Search is the process of finding the position (or location) of a given element (say x) in the linear array. The search is said to be successful if the given element is found in the array; otherwise unsuccessful.

A Binary search algorithm is a technique for searching the position of a particular value (say x) within a sorted array A. The real world example of the binary search is 'dictionary,' which we are using in our daily life to find the meaning of any word.

Given a sorted array of n elements, the fundamental idea of binary search is that the algorithm compares the value of input element x with the value of the mid element in the array. If the value matches with the middle element, it will return the index of the middle. Otherwise, if x is less than the middle element, then the algorithm recurs to the left side of middle element, else recurs for the right side of the middle position.

Algorithm

/* Input: A sorted (ascending) linear array of size n.

Output: This algorithm find the position of the search element x in linear array A. If search is successful it will return the location of the searched element x, otherwise returns -1, which indicates x is "not found". Here variables low and high are used for the first element and last element of the array to be searched, and variable mid is used as the index of the middle element of that array. */

```
BinarySearch_Iterative(A[1…n],n,x)

low=1

high=n

while(low<=high)

mid= (low+high)/2

if(A[mid]==x]

return mid; // x is found

else if(x

high=mid-1;

else low=mid+1;

return -1 // x is not found
```

Analysis: We know that any problem, which is solved by using Divide-and-Conquer having a recurrence relation of the form: T (n) = a*T (n/b) + f(n)

At each iteration, the array is divided into two sub-arrays, but we will consider only one sub-array in the next iteration. So the value of a=1 and b=2 and f(n)=k where k is a constant having value less than n.

Thus a recurrence relation for a binary search can be written as: T (n) = T (n/2) + k;

Using substitution method, to solve this recurrence relation, we have:

k+k+k+…… up to (logn) terms = k. (logn) = O(logn).

Best Case Analysis: The best case for binary search will be when the searched element x is found in the middle of the array. At that moment the complexity will be Θ(1).

Worst Case Analysis and Average Case Analysis: For this algorithm worst case will be when the searched element is not found in the array or element is at the end points and the average case will be when the element lies between one of the sub-arrays. Hence in both cases, complexity will be O(log n).

3.5 Finding Maximum and Minimum

To find maximum and minimum in a set of n elements, we have one straight forward strategy of comparing all elements to each other and find the desired output.

Algorithm: Maximum_Minimum

integer i, n;

max ← min← A.(1)

for i← 2 to n do

if A(i) >max

then max ←A(i) endif

if A(i) < min

then min - A (i) endif

repeat

However, this technique needs lots of iterations, i.e., may be, 2 n -2 if we apply divide and conquer technique on the same problem then we can divide the array into two sub- arrays, will find maximum and minimum of each sub-array and after comparing the outcomes of these two sub-arrays, we can get maximum and minimum of the complete array.

Algorithm:

MaxMin(I,j,fmax,fmin)

Int i,j;

global n, A(I:n)

case

1: i = j:fmax←fmin←A(i)

2: i = j - 1:ifA(i) < A(i) then fmax ← A(j); fmin←A(i)

Else fmax← A(i); fmin← A(j)

endJf

3: else: mid $\leftarrow \lfloor (i+j)/2 \rfloor$

call MAXMIN(i, mid, gmax, gmin)

call MAXMIN(mid + I,j, hmax, hmin)

fmax←max(gmax, hmax)

fmin← min(gmin, hmin)

end case

end MAXMIN

Analysis: Here the array is divided into two sub-array of almost same size. Hence recurrence relation is given by

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2$$

In this recurrence the term $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$

comes from conquering the two sub-problems, where we divide the original recurrence.

Clearly, by induction, we get, T (n)=3 n /2-2, which is 25% lesser than the previous method.

⇒ T (n)=O(n)

Best Case Analysis: When we have two elements in the array i.e. n =2⇒ T (2)=1⇒ T (n)=Θ(1)

Average case and Worst Case Analysis: For n>2, we will have both cases and complexity will be, T(n)=O(n).

3.6 Merge Sort

Merge sort is a recursive algorithm that ceaselessly splits a list into two halves. If the list is empty or has only one item, it is already sorted by definition (the base case). If the list has only one item then we need not to compare it otherwise, we split the list and recursively invoke the method of merge sort on both halves. Once both the halves are sorted, the essential operation, called a merge , is performed. Merging is the way towards taking two smaller sorted lists and joining them together into a single, sorted, new list.

Application: In the real world scenario, it is used in the sensitive analysis of Google ranking function, Rank aggregation for meta-searching on the Web and so on. Here we can apply the divide-and-conquer strategy to sort a given sequence of data items, with the help of following steps:

Recursively split the list into two halves (i.e., into subsequences) until we reach at the point where every subsequence contains just a single data item (i.e., singleton subsequence)

Now, recursively combine these subsequences back, preserving their required order (i.e., ascending or descending order).

Algorithm for main function:

MERGESORT(A,p,r): // A is the array, p is the indices of first element and r that of last element

if (p

then q ← □p+r/2□

MERGESORT(A,p,q)

MERGESORT(A,q+1,r)

MERGE(A,p,q,r)

Algorithm for function Merge(A, p, q, r) :

MERGE (A , p , q , r)

n 1← q - p +1

n 2← r - q

Create arrays L[1 .. n 1+ 1] and R[1 .. n 2 + 1]

FOR i ← 1 TO n 1

DO L[i] ← A[p + i - 1]

FOR j ← 1 TO n 2

DO R[j] ← A[q + j]

L[n 1 + 1] ← ∞

R[n 2 + 1] ← ∞

i ← 1

j ← 1

FOR k ← p TO r

a. DO IF L[i] ≤ R[j]

I THEN A[k] ← L[i]

II i ← i + 1

b. ELSE A[k] ← R[j]

I j ← j + 1

Analysis of Merge Sort:

For n = 1, time is constant and given by T(1)=1.

Otherwise,

Time is taken by merge sort for n elements = 2 z*time taken by merge sort for n/2 elements + time taken to merge two arrays, each containing n/2 elements.

Time to merge two arrays of each n/2 elements = n.

$$\Rightarrow T(n) = 2T\left(\frac{n}{2}\right) + n \text{ Where } T(n) = \begin{cases} 2T^{1}\left(\frac{n}{2}\right) + n & \text{if } n = 1 \\ \text{otherwise}' \end{cases}$$

Solving this equation using recurrence method, we get complexity of merge sort = O (nlogn)

Example of Merge Sort:

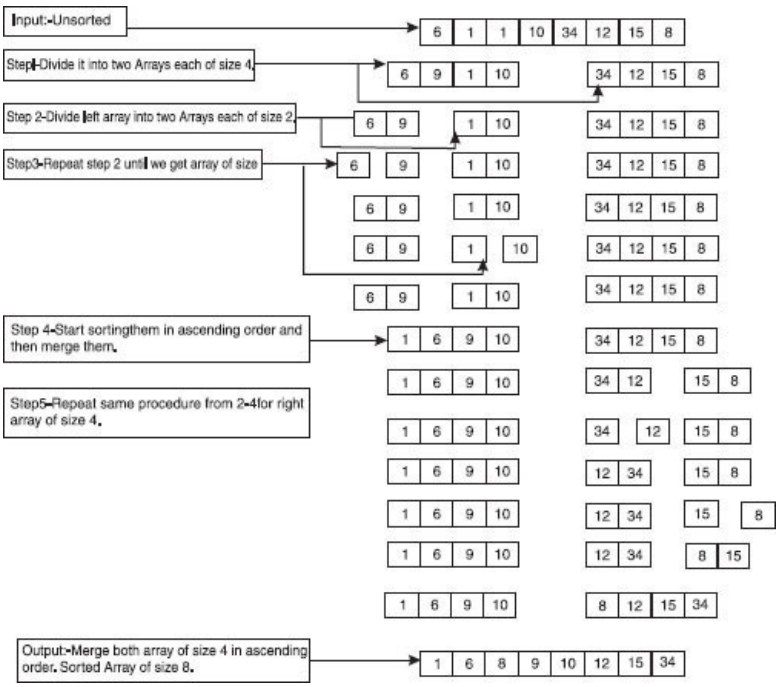


Figure: Merge Sort Using Divide and Conquer

3.7 Quick Sort

This algorithm was designed by C. A. R. Hoare in 1962. Quick sort also uses the Divide and conquer strategy but doesn't need an extra space of O (n) that is required by merge sort to combine that array. In this algorithm, we choose one element as the pivot and partition the array into two sub-arrays. By comparing with this pivot, we arrange elements of these sub-arrays.

All elements, less than pivot arrange in left sub-array and

All elements, greater than pivot arrange in the right sub-array.

Algorithm

QUICKSORT(A, p, r)

if p < r

then q ← PARTITION(A, p, r)

QUICKSORT(A, p, q - 1)

QUICKSORT(A, q + 1, r)

To sort the array A using quick sort, the initial call will beQUICKSORT(A, 1, length[A]).

PARTITION(A, p, r)

x ← A[r]

i ← p - 1

for j ← p to r - 1

do if A[j] ≤ x

then i ← i + 1

exchange A[i] ← A[j]

exchange A[i + 1] ← A[r]

return i + 1

Analysis of Quick Sort: The run time of quick sort algorithm depends on whether the partitioning is balanced or unbalanced, and this, in turn, depends on the choice of portioning element. If the partitioning is balanced, the algorithm runs asymptotically as fast as the algorithm for merge sort, otherwise, it can run asymptotically as slowly as the algorithm for insertion sort. Let us assume that the pivot divide array in two sub-arrays of (n - k) and k .

That's why total time, T (n)= T (k)+ T (n - k)+ c * n where c is a constant and n is the time required in rearranging array elements.

Best Case analysis: When input array is not sorted then the partition produces two sub-arrays, each of size not more than n /2. Number of elements may be □ n /2□ and □ n /2□.

$$\Rightarrow T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c * n$$
$$\Rightarrow T(n) = 2T(n/2) + c * n$$

By solving this using some recurrence relation method, we will get T (n)=O(n log n)

Worst Case Analysis : On the other hand; consider choosing the absolute minimum of each array as a pivot, i.e., k =1 and n - k = n -1. Solving this we will get T (n)=O(n ²)

Average Case Analysis: Suppose input array is divided into the randomized size of two sub-arrays of size i and n-i.

T (n)= T (k)+ T (n - k)+ c * n

Solving this again we will get T (n)=O(n log n)

Example of Quick Sort:

Figure: Quick Sort Using Divide & Conquer

3.8 Select K th Smallest Element

This algorithm find out the kth smallest element in the array such that, $1 \le k \le n$. The remaining elements are rearranged in such a manner that $A(k) = t$, $A(m) \le t$ for $1 \le m \le k$ and $A(m) \ge t$ for $k \le m \le n$.

Algorithm

function SELECT(A, n, k)

//Within the array A(1… n) the k th smallest element s is found and placed at position k.

//It is assumed that $1 \le k \le n$. The remaining elements are rearranged in such a manner that $A(k) = t$, // $A(m) \le t$ for $1 \le m \le k$. and $A(m) \ge t$ for $k \le m \le n$

integer n, k, m, r,j;

$m \leftarrow 1$; $r \leftarrow n + 1$;

loop

a. j ←PARTITION (A,m, j) // this partition function is same as used in Quick Sort case

1: k = j: return

2: k < j: r ← j

3: else: m ← j +1

b. encase

repeat

endSELECT

Analysis : The run time of Selection algorithm depends on the partitioning; whether it is balanced or unbalanced, and this depends on which elements are used for partitioning. Let us assume that the pivot divide array in two sub-arrays of (n - k) and k .

That's why total time T (n)= T (k)+ T (n - k)+ c * n , where c is a constant and n is time to rearrange array elements.

Best Case Analysis: In the best case, the list is already sorted. Now array will be divided into two equal sized sub-arrays, and only one of these sub-arrays is needed to be iterated.

$$\Rightarrow T(n) = T(\lfloor n/2 \rfloor) + c * n$$
$$\Rightarrow T(n) = T(n/2) + c * n$$
$$\Rightarrow T(n) = O(n)$$

Worst Case Analysis: In the worst case, the list is sorted in reverse order. Now on i th iteration, we will get the i th smallest element.

T (n) = T (n - 1) + O(n)

⇒; T (n) = O(n ²)

Average Case Analysis: Average case analysis will be same as of Quick sort.

3.9 Strassen's Matrix Multiplication

Given two square matrices Aand B of size n x n each, find their resultant multiplication matrix. Below is a way to multiply two matrices.

multiply(int A[][N], int B[][N], int C[][N])

for i← 0 to N

for j ← 0 to N

C[i][j]← 0

for k ← 0 to N

i. C[i][j] += A[i][k]*B[k][j];

The Time Complexity for this brute force method is O(N ³).

A. Divide and Conquer method for matrix multiplication-1)

1) Divide matrices A and B in four sub-matrices each of size N/2 × N/2 as shown in the below diagram. 2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.

A,B and C are square matrices of size N×N

a, b, c and d are submatrices of A, of size N/2×N/2

e, f, g and h are submatrices of B, of size N/2×N/2

In the above method, we perform 8 multiplications for matrices of size N/2 × N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the recurrence relation for divide and conquer method can be written as:

$T(N) = 8T(N/2) + O(N^2)$

From Master's Theorem, the time complexity of above relation is $O(N^3)$ which is unfortunately same as of the above naive method.

B. Strassen's method reduces the number of recursive calls for the same matrices to only seven calls . Strassen's method is very similar to above discuss divide and conquer strategy, that this algorithm also divide matrices to sub-matrices of size N/2 × N/2 as shown in the above diagram, but in Strassen's method, the multiplication of four sub-matrices are calculatedly using following formulae.

A,B and C are square metrices of size N×N

a, b, c and d are submatrices of A, of size N/2×N/2

e, f, g and h are submatrices of B, of size N/2×N/2

p1, p2, p3, p4, p5, p6, and p7 are submatrices of size N/2 ×N/2

Time Complexity of Strassen's Method

Addition and Subtraction of two matrices take $O(N^2)$ time. So recurrence relation for Strassen's method can be written as

$T(N) = 7T(N/2) + O(N^2)$

From Master's Theorem, time complexity of Strassen's method will be $O(N \text{ Log}7)$ which is approximately $O(N^{2.8074})$

In practice, strassen's Method is not preferred for the following reasons. 1) The constants used in Strassen's method have high values, and for a normal application, Naive method works better. 2) For Sparse matrices, better methods exist which are specially designed for them. 3) The dividing matrix into sub-matrices in recursive calls takes extra space. 4) Because of the limited precision of computer arithmetic on non- integer values, larger errors accumulate in Strassen's algorithm in comparison to simple matrix multiplication method.

3.10 Convex Hull

A polygon is convex if any line segment joining two points on the boundary remains inside the polygon. Equivalently, if you stroll around the boundary of the polygon in the counter clockwise way you take left turns.

The convex hull is a set of points in the plane is the smallest convex polygon for which each point either resides on the boundary or on inside of the polygon. One may think about the points as being nails sticking out of a wooden board: then the convex hull is the shape framed by a tight elastic band that encompasses every one of the nails. A vertex is a corner of a polygon.

Here two algorithm Graham's scan and Jarvis march that computes the convex hull of a set of n points in counter clockwise order. Several methods that compute convex hulls include the following.

3.10.1 Graham's Scan

Here in this strategy, recognize one vertex of the convex hull and sort the alternate points as viewed from that vertex. Then the points are scanned in order. Let x 0 be the furthest left point (which is ensured to be in the convex hull) and number the rest of the points by the angle from x 0 going counter clockwise: x 1 , x 2… x n-1 . Let x n = x 0 , the picked point. Assume that no two points have the similar angle from x 0 . The algorithm is easy to state with a single stack

Algorithm:

Sort points by angle from x 0

Push x 0 and x 1 into the stack. Set i=2

While i ≤ n do:

If x i makes left turn w.r.t. top 2 items on the stack

I then pushxi;

Il i++

else pop and discard

Analysis: Each time the while loop is executed; a point is either stacked or discarded. Since a point is looked at just once, the loop is executed at most 2n times. There is a constant-time subroutine for checking, given three points altogether, regardless of the angle is a left or a right turn. This gives an O(n) time algorithm, aside from the initial sort which requires time O(nlog n).

3.10.2 Jarvis March

This is also called the wrapping algorithm . This algorithm finds the points on the convex hull in the order in which they show up. It is fast if there are just a couple of points on the convex hull, however moderate if there are many.

Let x 0 be the furthest left point. Let x 1 be the first point counter clockwise when seen from x 0 . At that point x 2 is the first point counter clockwise when seen from x 1 , and so on.

Algorithm:

i = 0

while not done do

x i+1 = first point counter clockwise from x i

Analysis: While loop is executed h times where h is the number of vertices on the convex hull. So Jarvis March takes time O (n h).

Best case analysis: When h = 3

Worst case analysis: When n = h

3.10.3 Divide and Conquer Method

This is one of the popular techniques to find convex hull. In this strategy, we divide the set into two halves and then find the convex hull of each set.

Algorithm:

Break the n points into two halves

Find convex hull of each subset.

Combine the two hulls into the overall convex hull.

Here the second step is a recursive call of two subsets. If a point lies inside the overall convex hull, then it must be inside the convex hull for any subset of points that contain within it. So the task is for given two convex hulls find the union of their convex hull.

3.10.4 Joining Two Hulls

It helps to find the convex hulls that do not overlap. To confirm this, all the points are presorted from left to right side. Then we have a left and right convex hull. Afterward define a bridge, i.e., aline segment, joining a vertex on the left convex hull and a vertex on the right convex hull such that this line segment does not cross the boundary of either polygon. For this purpose, we need the upper and lower bridges.

3.10.4.1 Steps to Defines the Bridge:

Start with any bridge. A bridge is confirmed if you join the furthest right vertex on the left side of hull to the furthest left vertex on the right side of the hull.

Keep the left end of the bridge fixed; check if the right edge can be raised. Check for the next vertex on the right polygon moving clockwise and check whether that would be a bridge. Otherwise, check if the left edge can be uplifted while the right end remains fixed.

If it makes no progress in step 2 (cannot raise either side), then stop else repeat step 2.

Analysis: The key is to perform step 2 in constant time. For this, each vertex must have a pointer to the next vertex going clockwise and going counter clockwise. The total work done for combining two sets is proportional to the number of vertices. It means that the overall algorithm takes time O(nlog n).

3.11 Summary

Algorithm	Recurrence Relation	Time Complexity		
		Worst-case	Best-case	Average-case
Binary Search	Worst Case: $T(n) = T(n/2) + k$	$O(logn)$	$\Theta(1)$	$O(logn)$
Max & Min	Worst Case & Avg Case: $T(n)=3n/2-2$	$O(n)$	$\Theta(1)$	$O(n)$
Merge sort	Best Case & Worst Case: $T(n) = 2T(n/2) + O(n)$	$\Theta(nlgn)$	$\Theta(nlgn)$	$\Theta(nlgn)$
Quicksort	Best Case: $T(n) = 2T(n/2) + O(n)$ Worst Case: $T(n) = T(n-1) + O(n)$ Average Case: $T(n) = T(3n/8) + T(7n/10) + O(n)$	$\Theta(n^2)$	$\Theta(nlgn)$	$\Theta(nlgn)$
Strassen's Matrix Multiplication	Worst Case: $T(n) = 7T(n/2) + O(n2)$	$O(n^{\log 7}) \approx O(n^{2.81})$		

REVIEW EXERCISE

What is divide and conquer strategy? Explain with suitable example.

Explain iterative as well as the recursive solution for the binary search method? Define its complexity also.

Write a program for recursively binary search.

Devise a binary search program that doesn't divide the array into two sub- arrays of equal size, but it divides it into one-third and two-third part. Calculate its time complexity.

Perform dry run for Quick Sort using array {8, 9, 2, 1, and 4}.

Draw the tree for the given sequence using the recursive calls of merge sort. 9,4,2,11,6,5,15,7

Perform dry run for Merge Sort using array A= {65, 70, 30, 25, 50, and 40}.

MULTIPLE CHOICE QUESTIONS & ANSWERS

You have an array of size n. Assume you implement quick sort by always picking the middle element of the array as the pivot. Then the tightest upper bound for the worst case performance is given by

(a) O(n ²)

(b) O(nLogn)

(c) O(nLogn)

(d) O(n ³) [GATE 2014]

Ans. (a) O(n ²)

The middle element will always be an extreme element (minimum or maximum) in sorted order.Therefore time complexity in worst case scenario will be O(n ²)

Let A be a square matrix of order n x n. Consider the following program code. What will be the expected output?

C= 100

For i ← 1 to n do

For j ← 1 to n do

{temp = A[i][j]+C

A[i][j] = A[j][i]

A[j][i]=temp - C}

For i ← 1 to n do

For j ← 1 to n do

Print A[i][j]

(a) The matrix A

(b) Transpose of matrix A

(c) Adding 100 to the upper diagonal elements and subtracting 100 from the lower diagonal elements of A

(d) None of the above [GATE 2014]

Ans. (a) The matrix A

If we look the execution of inner statements of first loops, we can notice the swapping of A[i][j] and A[j][i] for all i and j. Since the loop runs for all elements of the array, every element A[l][m] would be swapped twice, once for i = l and j = m and then for i = m and j = l. Swapping twice a single element means the matrix will not change.

Consider the following program code. What will be the expected outcome?

int main()

{

int x,y,n,m;

scanf("%d %d", &x, &y);

m= x; n=y;

while(m !=n)

{if (m>n)

m= m-n;

else n=n-m;}

printf("%d",n);} (GATE 2004)

(a) x + y using repeated subtraction

(b) x mod y using repeated subtraction

(c) the greatest common divisor of x and y

(d) the least common multiple of x and y

Ans. (c) the greatest common divisor of x and y

Explanation: This is the code for of Euclid's algorithm to find GCD

Consider the polynomial equation of the form p(x) = a0 + a1x + a2x2 +a3x3, where ai != 0, for all i. The minimum number of multiplications required to evaluate p is:

(a) 3

(b) 4

(c) 6

(d) 9

Ans. (a) 3

Explanation: Number of multiplications can be minimized by using following order for parenthesis of the given expression.

$p(x) = a^0 + x (a^1 + x (a^2 + a^3 x))$

Suppose in C programing, you are provided with the following function declaration

int partition (int a[], int n);

The function treats the first element of array a as a pivot, and perform the quick sort to arrange smallest elements in the left part and largest elements in right part of array. The following C programming function is used to find the k th smallest element in an array a[] of size n using the above declared partition function. We assume k ≤ n

intk_smallest (int a[], int n, int k)

{intleft_end = partition(a,n);

if (left_end+1==k)

{return a[left_end];}

if (left_end +1 > k)

{returnk*smallest(*___);}

else {return k*smallest(*___);}

}

The missing argument lists are respectively

(a) (a, left*end, k) and (a+left*end+1, n-left*end-1, k-left*end-1)

(b) (a, left*end, k) and (a, n-left*end-1, k-left_end-1)

(c) (a, left*end+1, N-left*end-1, K-left*end-1) and(a, left*end, k)

(d) (a, n-left*end-1, k-left*end-1) and (a, left_end, k)

Ans. (a) (a, left*end, k) and (a+left*end+1, n-left*end-1, k-left*end-1)

Searching an element x in an array ‘arr[]’ of size n, can be solved in O(Logn) time if.

(1) Array is already sorted

(2) Array is already sorted and rotated by k, where value of k is given to you and k <= n

(3) Array is already sorted and rotated by k, where value of k is NOT given to you and k <= n

(4) Array is not sorted

(a) Option 1 Only

(b) Option 1 & 2 only

(c) Options 1, 2 and 3 only

(d) All Options

Ans. (c) Options 1, 2 and 3 only

Without using in-built function to calculate power (pow() function in C), if you need to calculate x^n where x can be any number and n is a positive integer. What can be the possible best case time complexity for this function?

(a) O(n) (b) O(nLogn)

(c) O(LogLogn)

(d) O(Logn)

Ans. (d) O(Logn)

Explanation: We can calculate x^n using divide and conquer in order of O(Logn) time.

C HAPTER -4

Greedy Algorithm

In this chapter student will understand:

What is greedy algorithm? What are its advantages? What are the area where we can apply greedy algorithms and how we optain optimum result from this?

4.1 Optimization Problem

An optimization problem comprises a set of constraints and optimization function. A solution that satisfies all constraints is a feasible solution, and a feasible solution with the best possible value of optimization function is an optimal solution .

4.2 Greedy Algorithm

Greedy Algorithms are used to optimize a problem in which a set of choices that may refer as a feasible solution is made to arrive at an optimal solution. The idea of the greedy algorithm is to make each choice in a local optimal manner just like the thought of a greedy person whose interest is only in getting maximum benefit without considering whether the benefit will sustain for a long time or not.

The greedy method suggests that one can devise an algorithm which works in stages, considering one input at once. At each stage, a choice is made with respect to regardless of whether a specific input is considered for an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will bring about an infeasible solution, at that point this input is not added to the partial solution.

The advantage of this approach is that the solutions to smaller sub-problems are easy to understand. The disadvantage of this approach is that the short term optimal solution can result in a worst case long term solution. For example: the Greedy algorithm is often used in an ad-hoc mobile network to route a packet, choosing bus for given route etc.

Algorithm

Procedure GREEDY(A.n)

//A(I:n) contains then inputs//

solution ←ϕ / /initialize the solution to empty //

fori ←1 to n do

x ←SELECT(A)

if FEASIBLE(solution,x)

then solution ←UNION(solution,x)

endif

repeat

return (solution)

end GREEDY

Where Select function is used to select an input from A and assign it to x. Feasible(), always gives some Boolean expression finding whether x can

be included in set of optimum solution or not.

4.3 Coin change problem

Greedy algorithm always gives a coin of highest denomination and then does for the remaining amount.

For example, if we have coin set of (1,2,5,10,20}, to make sum of 40.

We have solution {20,20} and it may be {10, 10, 10, 10}.

Here constraint is 40.

Optimization function is a sequence of coins i.e. 2 & 4.

So optimal solution is {20,20}

Again the optimal solution for this will be same.

But this not always happen that a greedy algorithm will give an optimal solution.

Suppose we have coins of {10, 40, 60} and to make a sum of 80. We may have following options:

Using greedy algorithm we first take 60 ≤ 80 and then 60+2*10=80, However the optimal solution for this is= 40+40.

4.4 Activity selection problem

In daily life if we have a large number of task to perform than we have to schedule them to complete them within a given time period. Same is the case is with computer, that if multiple user submit many activities for processing then the system has to choose them on the bases of some scheduling.

Activity selection problem handles all such kind of issue to get a optimal result in optimum time. Suppose we have a set A = {A1, A2, …, An} of n activities with their start and finish times (si, fi), such that 1 ≤ i ≤ n. We have to select maximum set S of “non- overlapping” activities to get optimum result. We are scheduling here to get largest set of disjoint activities.

Sort activity by finish time (let A1, A2, …, An denote sorted sequence)

Pick the first activityA1from the sequence

Remove all activities having start time before the finish time of A1

Recursively solve problem on remaining activities.

Algorithm

RECURSIVE-ACTIVITY-SELECTOR(s, f, i, j)

m ← i + 1

while m < j and $s_m < f_i$

Find the first activity in S ij .

do m ← m + 1

if m < j

then return {a m }

RECURSIVE-ACTIVITY-SELECTOR(s, f, m, j)

else return ∅

Example: We have 10 activities sorted according to their finish time (0,2), (1,3), (2,4), (3,5), (0,6),(4,7),(3,8),(5,9),(7,10), (2,11)

The largest set of activities will be job1, job3, job6, job9.

Analysis: The sorting of jobs take time=O(n log n)

Activity selector take time = Θ(n)

The complexity of activity selection problem =O(n log n)

Note: No overlapping activities mean, no two activities can share same resources. We say two activities are compatible, if their time periods are disjoint.

4.5 Knapsack Problem

Suppose a thief decided to steal from a shop having a bag (knapsack) of capacity M. Thief enters a store. The store is having i objects where weight of each object is represented by wi . Any number of objects can be put into knapsack as long as the weight limit M is not exceeded.

Input- n objects and a knapsack of capacity M.

Each object ihas weight wi and value pi.

Fraction of object is xi , $0 \le xi \le 1$ yields a profit of pi . xi.

Our objective is fill that knapsack.

The optimization function says:

I. $\text{Maximize} \sum_{i=1}^{n} p_i . x_i$

II. $\sum_{i=1}^{n} w_i x_i \le M$

III. and $0 \le xi \le 1$ where $1 \le i \le n$

Where p and w are positive integers.

A solution satisfying condition II and III is called a feasible solution and which satisfies condition I is called a optimal solution for the given set of objects and knapsack.

Algorithm

procedure GREEDY _KNAPSACK(P, W. M, X, n)

//P(1 :n) and W(1 :n) contain the profits and weights respectively of all the objects, ordered such that // P (i)/ W (i)≥ P (i +1)/ W (i +1) . M is the knapsack size and X(1:n) is the solution vector.

real P(1:n), W(1:n), X(1:n), M, cu;

integer i, n ;

X ← 0 // initialize solution to zero//

cu ← M //cu = remaining knapsack capacity//

for i ← 1 to n

do if W(i) > cu

then exit

endif

X(i) ← 1

cu ← cu - W(i)

repeat

a. if i ≤ n

b. then X(i) ← cu/W(i)

c. endif

end GREEDY _KNAPSACK

4.5.1 Fractional knapsack

If we are using the fractional amount of any object to fill the knapsack then it is an example of fractional knapsack.

Example: Supposes we have three objects of weight (18, 15, 10) and value associated with them are (25, 24, 15) and size of knapsack is 20.

n=3

(w1, w 2 , w 3) = (18, 15, 10)

(p1, p2, p3) = (25, 24, 15)

Largest profit strategy: In this strategy we always pick objects with highest profit if the weight of object exceeds with the knapsack capacity which fill it with the fraction of it.

Here in this example we take object 1 first. x 1 =1

P=25, remaining capacity, C = 20-18=2

Now pick object 2 as weight of object 2 is greater than remaining capacity of knapsack take fraction of it. x 2 =2/15

P=25+2/15*24=28.2

Since knapsack is full now, x 3 =0

The feasible solution is (1, 2/25, 0)

Smallest weight strategy: In this strategy, don't fill knapsack quickly. We always pick objects with lowest weight if the weight of object exceeds with the knapsack capacity which fill it with the fraction of it.

Here in this example we take object 3 first. x 3 =1

P= 15, C = 20-10=10

Now pick object 2 as weight of object 2 is greater than remaining capacity of knapsack which take fraction of it. x 2 = 10/15=2/3

P=15+24*2/3=31

Since knapsack is full now, x 1 =0

The feasible solution is (0, 2/3, 1)

Largest profit weight ratio strategy: In this strategy we always pick objects with highest profit weight ratio, if the weight of object exceeds with the knapsack capacity which fill it with the fraction of it.

P 1 / w 1 = 25/18=1.389

P 2 / w 2 = 24/15=1.6

P 3 / w 3 = 15/10=1.5

Here in this example we take object 2 first. x 2 =1

P=24, C=20-15=5

Now pick object 3 as weight of object 3 is greater than remaining capacity of knapsack take fraction of it. x 3 = 5/10=1/2

P=24+15*1/2=31.5

Since knapsack is full now, x 1 =0

The feasible solution is (0, 1, 1/2)

4.5.2 0/1 knapsack

If to fill knapsack we are using complete object or let this knapsack be unfilled, then it is called 0/1 knapsack.

Example: In same example which we have discussed in case of fractional knapsack, we have three objects of weight (18, 15, 10) and value associated with them are (25, 24, 15) and the size of knapsack is 20.

n=3

(w1, w 2 , w 3) = (18, 15, 10)

(p1, p2, p3) = (25, 24, 15)

Largest profit strategy: In this strategy we always pick objects with highest profit if the weight of object exceeds with the knapsack capacity don't include it.

Here in this example we take object 1 first. x 1 =1

P=25, remaining capacity, C = 20-18=2

Now we can't pick any object for remaining capacity i.e. 2 as weight of object 2 is greater than remaining capacity of knapsack.

The feasible solution is (1, 0, 0)

Smallest weight strategy: In this strategy, don't fill knapsack quickly. We always pick objects with lowest weight if the weight of object exceeds with the knapsack capacity don't try to include it with the fraction of it.

Here in this example we take object 3 first. x 3 =1

P= 15, C = 20-10=10

Now pick object 2 as weight of object 2 is greater than remaining capacity of knapsack, so don't include it.

The feasible solution is (0, 0, 1)

Largest profit weight ratio strategy: In this strategy we always pick objects with highest profit weight ratio, if the weight of object exceeds with the knapsack capacity don't fill with fraction.

P 1 / w 1 = 25/18=1.389

P 2 / w 2 = 24/15=1.6

P 3 / w 3 = 15/10=1.5

Here in this example we take object 2 first. x 2 =1

P=24, C=20-15=5

Now pick object 3 as weight of object 3 is greater than remaining capacity of knapsack leave it unfilled.

The feasible solution is (0, 1, 0)

Analysis: If the jobs are sorted according to pi / wi , and then loop will take O(n) time to execute. However if objects are not sorted then time will be O(n log n).

4.6 Job sequencing problem with deadlines (task scheduling algorithm)

We are given a set of 'n' jobs. Related with each job there is a integer dead line di≥0 and a profit pi>0. For any job i the profit pi is earned if and only if the job is completed by its dead line. To finish a job one need to process the job on a machine for one unit of time Just a single machine is available for processing the jobs. A feasible solution for the problem will be a subset 'j' of jobs with the end goal that each job in this subset can be finished by its deadline. The value of a feasible solution 'j' is the sum of the profits of the jobs in 'j'. An optimum solution is a feasible solution with maximum value.

The problem involves identification of a subset of jobs which can be completed by its dead- line. Therefore the problem suites the subset methodology and can be comprehended by the greedy method.

Example: Obtain the optimal sequence for the following jobs.

		j1 j2 j3 j4
(P1, P2, P3, P4)	=	(100, 10, 15, 27)
(d1, d2, d3, d4)	=	(2, 1, 2, 1)

n =4

Step 1: Arrange profit Pi in descending order and also arrange corresponding deadline.

(P1, P2, P3, P4)	=	(100, 10, 15, 27)
(d1, d2, d3, d4)	=	(2, 1, 2, 1)

Step 2: Now make pairs of {no. of jobs, maximum deadline} to find how many jobs pairing can be done.

⇨ Min{4,2} =2

⇨ A combination of 2 jobs can be made at maximum as each job take a unit time.

Step 3: Select jobs on the basis of timeline for example in the above problem, first choose the job with less deadline value, here '1' is selected and then deadline value '2' is selected.

Feasible solution	Processing sequence	Value
(1, 2)	(2,1)	100+10=110
(1,3)	(1,3) or (3,1)	100+15=115
(1,4)	(4,1)	100+27=127
(2,3)	(2,3)	10+15=25
(3,4)	(4,3)	15+27=42
(1)	(1)	100
(2)	(2)	10
(3)	(3)	15
(4)	(4)	27

In this example, solution '3' is the optimal. In this solution only jobs 1&4 are handled and the value is 127. These jobs must be handled in the order j4 took after by j1. The process of job 4 begins at time 0 and ends at time 1. And the processing of job 1 begins at time 1 and closures at time.

Therefore both the jobs are completed within their deadlines. The optimization measure for determining the next job to be selected for the final solution is as per the profit. The next job to incorporate is what increments ∑pi the most, subject to the constraint that the resulting “j” is the feasible solution. Consequently the greedy strategy is to consider the jobs in decreasing order of profits.

Algorithm

GreedyJob (d,j,n)

//j is a set of jobs that can be completed by their dead lines

j = {1};

for i: = 2 to n do

if (all jobs in j □{i} can be completed by their dead lines) then

j: = j □{i};

end if

repeat

endGreedyjob

Analysis : Sorting and selection of jobs will take O(n log n)time. Inserting jobs into partial solution will take O(n) time and checking whether the new solution is feasible or not takes O(n) time. Hence worst case complexity will be O(n ²).

REVIEW EXERCISE

What is activity selection problem? Mention few of its applications?

Compare 0/1 and fractional knapsack methods to get the best optimal solution to a problem?

What is the difference between greedy and divide and conquer algorithm?

Discuss job sequencing problem with its deadline in detail.

MULTIPLE CHOICE QUESTIONS & ANSWERS

There are n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time. Consider the following 6 activities.

start[] = {1, 3, 0, 5, 8, 5};

finish[] = {2, 4, 6, 7, 9, 9};

Ans. The maximum set of activities that can be executed by a single person is {0, 1, 3, 4}

Explanation: The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity. For this perform following steps:

Sort the activities according to their finishing time

Select the first activity from the sorted array and print it.

Do following for remaining activities in the sorted array.

a) If the start time of this activity is greater than the finish time of previously selected activity then select this activity and print it.

Using Dijkstra’s single source shortest-path algorithm on the following edge weighted directed graph with vertex P as the source. In what order do the nodes get included into the set of vertices for which the shortest path distances are finalized? (GATE CS 2004)

(a) P, Q, R, S, T, U

(b) P, Q, R, U, S, T

(c) P, Q, R, U, T, S

(d) P, Q, T, R, U, S

Ans. (b)

Suppose the message contains the following characters with their frequency, using the compression technique used in Huffman Coding, how many bits will be saved in the encoded message?

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

(a) 224

(b) 800

(c) 576

(d) 324

Ans. (c)

Explanation:

Total number of characters in the message = 100.

Each character takes 1 byte.

So total number of bits needed = 800.

After Huffman Coding , the characters can be represented with:

f: 0

c: 100

d: 101

a: 1100

b: 1101

e: 111

Total number of bits needed = 224

Hence, number of bits saved = 800 - 224 = 576

Consider the undirected graph below:

Using Prim's algorithm to construct a minimum spanning tree starting with node A, which one of the following sequences of edges represents a possible order in which the edges would be added to construct the minimum spanning tree?

(a) (E, G), (C, F), (F, G), (A, D), (A, B), (A, C)

(b) (A, D), (A, B), (A, C), (C, F), (G, E), (F, G)

(c) (A, B), (A, D), (D, F), (F, G), (G, E), (F, C)

(d) (A, D), (A, B), (D, F), (F, C), (F, G), (G, E)

Ans. (d)

Suppose the letters a, b, c, d, e, f have probabilities 12, 14, 18, 116, 132, 132, respectively.

Which of the following is the Huffman code for the letter a, b, c, d, e, f? (GATE - 2007)

(a) 0, 10, 110, 1110, 11110, 11111

(b) 11, 10, 011, 010, 001, 000

(c) 11, 10, 01, 001, 0001, 0000

(d) 110, 100, 010, 000, 001, 111

Ans. (a) 0, 10, 110, 1110, 11110, 11111

In a given string the letters a, b, c, d, e, f have probabilities 1/2, 1/4, 1/8, 1/16, 1/32, 1/32 respectively. Which of the following is the Huffman code for the letter a, b, c, d, e, f?

(a) 0, 10, 110, 1110, 11110, 11111

(b) 11, 10, 011, 010, 001, 000

(c) 11, 10, 01, 001, 0001, 0000

(d) 110, 100, 010, 000, 001, 111 [GATE 2007]

Ans. (a) 0, 10, 110, 1110, 11110, 11111

Explanation: After applying Huffman Coding Algorithm, We will get the following Huffman Tree. Here, we keep the least probable characters as low as possible by picking them first.

The letters a, b, c, d, e, f have probabilities 1/2, 1/4, 1/8, 1/16, 1/32, 1/32 respectively.

For i = 3, p =(5 * 12 + 98) mod 13 = 2 and
t_0 = (5 * 11+ 99) mod 13 = 11
For i = 4, p =(5 * 2 + 98) mod 13 = 4 and
t_0 = (5 *11+ 100) mod 13 = 12
for s ← 0 to 12 – 4 = 8
for the s = 0, p = 4 and t0 = 12
Hence t 1 = (5 (12 – ascii(T[1]) 5) + ascii(T [5])) mod 13
= (102 (12 – 98*5) + 98) mod 13 = 5
For s = 1, p=4 and t1 = 5
Hence t2 = (5 (5 – 99*8) + 98) mod 13
= 2
For s= 2, p = 4 and t2 = 2
Hence t3 = (5 (2 – 100*8) + 99) mod 13 = 4
Here t3 = = p = 4

C HAPTER -5

Dynamic Programming

In this chapter student will understand:

What is the dynamic method for problem-solving? How is it different from other strategies? Which are the main areas solved with dynamic programming? How is it different with the greedy method?

5.1 Dynamic Programming

Dynamic programming usually applies to optimization problems in which a set of choices must be made to arrive at an optimal solution. As the choices are made, subproblems of the form of actual problem often arise. Dynamic programming is effective when a given sub problem may arise from more than one partial set of choices; the basic idea is to store the solution to each such sub-problem so that we can refer this if the sub-problem will reappear.

Unlike in divide-and-conquer algorithms, partition the problem into a set of independent sub problems, solve the subproblems recursively, and then combine their solutions to find solution of the original problem. In contrast, dynamic programming is applicable when the sub-problems are not independent, when sub-problems share same sub-sub-problems. A dynamic-programming algorithm solves every sub-sub-problem just once and then saves its answer in a table, thereby avoiding the work of recalculating the answer every time the sub-sub-problem is encountered.

The development of a dynamic-programming algorithm is performed into a sequence of four steps:

Characterize the structure of an optimal solution.

Recursively defines the value of an optimal solution.

Calculate the value of an optimal solution in a bottom-up fashion.

Construct an optimal solution from computed information.

5.2 Matrix Chain Multiplication

Using this algorithm, for a given chain of matrices, our aim is to find the most efficient way to multiply these matrices together in minimal time. The problem is not actually to perform the multiplications, but to decide the order of the multiplications, which in turns results in minimum computation.

We have many ways to multiply a chain of matrices because matrix multiplication is associative. In other words, we may say that no matter in what order, we parenthesize the matrices, the result will remain same. For example, if we have four matrices A, B, C, and D, we could parenthesize them as follows:

(ABC)D = (AB)(CD) = A(BCD) = ….

However, the order in which we parenthesize the product, affects the number of simple arithmetic operations needed to compute the product, or the efficiency of computation. For example, we have A is a matrix of order 10 × 30 matrix, B is of order of 30 × 50 matrix, and C is of size 50 × 60 matrix. Then,

(AB)C = (10×30×50) + (10×50×60) = 15000 + 30000 = 45000 operations

A(BC) = (30×50×60) + (10×30×60) = 90000 + 18000 = 108000 operations.

Clearly the first order of parenthesization requires less number of operations.

Count number of Parenthesization $P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{Else } n \geq 2 \end{cases}$

This is related to function in combinatory called Catalan number related to number of differential binary trees on n nodes.

$P(n)=C(n-1)$ & $C(n) = \frac{1}{n+1}\binom{2n}{n}$ by applying Stirling's formulae i.e.

$C(n) \in \Omega(4^n / n^{3/2})$

Optimal substructure (structure of an optimal paranthesization): A simple solution is to place parenthesis at all possible positions, calculate the cost of multiplication for each placement, compare the cost and return the minimum value. In a set of matrices of size n, we can place the first set of parenthesis in n-1 ways.

For example , if the given chain is having 4 matrices A,B,C,D, then there are 3 way to place first set of parenthesis: A (BCD), (AB) CD and (ABC) D. So when we place a set of parenthesis, we divide the problem into sub problems of smaller size i.e. now the chain has three matrices. Now, the problem has optimal substructure property and can be easily solved by using recursion.

Minimum number of multiplication needed to multiply a chain of size n = Minimum of all n-1 placements (these placements create sub problems of smaller size).

A recursive approach: Let m [i,j] be minimum number of scalar multiplications needed to compute matrix A[I,j].

$m[i,j] = \begin{cases} 0 & \text{if } i == j \\ \min_{i<=k<j} m[i,k] + m[k,j] + p_i p_k p_j & \text{if } i < j \end{cases}$

For storing values of k we define another array s[i,j]=k, where k is the position at which we split the product A i..j to get an optimal paranthesization. That is,

$s[i,j] = k$ such that $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j$.

Computing the optimal cost: The third stage of the dynamic programming paradigm is to construct the value of an optimal solution in a bottom-up fashion. In our problem the only tricky part is to arrange the order of parenthesis with minimal cost to compute the values (so that it will be available when we need it). In the process of computing m [i , j] we need to access the computed values m [i , k] and m [k + 1, j] for each value of k, where k lies between i and j .

Let T = j - i + 1 denote the length of the sub chain being multiplied. The sub chains of length 1 (m [i , i]) are trivial. Then we build up by computing the sub chains of length 2, 3, …, n . The final answer is m [1, n].

Algorithm

Matrix-Chain (array p [1 .. n], int n)

1. Array $s[1 .. n-1][2 .. n]$;
2. FOR $i \rightarrow 1$ TO n DO
   i. $m[i, i] = 0$; // initialize
      1. FOR $T \rightarrow 2$ TO n DO // T=length of sub chain
      2. FOR $i \rightarrow 1$ TO $n - T + 1$ do
         a. $j \rightarrow i + T - 1$;
         b. $m[i, j] = \infty$;
         c. FOR $k \rightarrow i$ TO $j - 1$ DO // check all splits
   i. $r = m[i, k] + m[k + 1, j] + p[i - 1] * p[k] * p[j]$;
   ii. IF ($r < m[i, j]$)
      1. $m[i, j] = r$;
      2. $s[i, j] = k$;

return m [1, n] and s ; // here m[1,n] is (final cost) and s is (splitting markers)

Constructing optimal solutions : This is the last step of the dynamic programming paradigm where we find an optimal solution from computed information at third step. The array s[i, j] can be used to extract the actual sequence.

Algorithm

PRINT OPTIMAL_SOLUTION(s, i, j)

If i==j

Print" A i "

Else

print "("

PRINT OPTIMAL_SOLUTION(s, i, s(i,j))

PRINT OPTIMAL_SOLUTION(s, s(i, j)+1,j)

Print")"

Analysis: The space complexity of this procedure is O (n^2). Since the matrix m and s require O (n^2) space.

Since, the three for-loops are nested, and each one of them iterates at most n times (that is to say indices T, i , and j takes on at most n - 1 values). Therefore, the running time of matrix-chain procedure is O(n^3).

Example: The initial set of dimensions is <5,4,6,2,7> means there are four matrices A1 of dimensions 5×4, A 2 of 4×6, A 3 of 6×2 and A 4 of 2×7.

For the matrix chain multiplication we have to computevalue of m [i,j] using the formulae

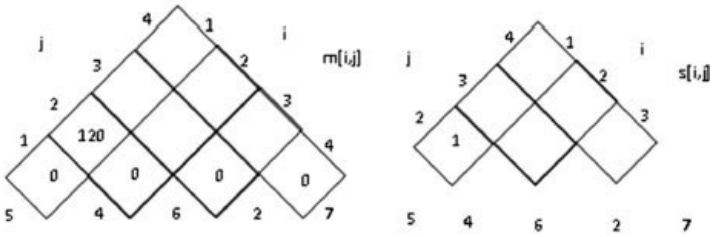
1. m [i , j] = min(m [i , k] + m [k + 1, j] + p i -1 p k p j And put the value of computed m [i,j] and s[i,j] = k in the matrix given below.

Step 1: Put m[1,1], m[2,2], m[3,3] and m[4,4]=0

Step2: $m[1,2] = \min_{1 \le k \le 2} (m[1,1] + m[2,2] + 4 \times 5 \times 6)$

$= m[1,1]+m[2,2]+120=0+0+120$

$= 120, k = 1$



Step 3: Similarly we will compute values of m [2, 3] and m [3, 4] which will be as follows

$$m[2,3] = \min_{2 \le k \le 3} (m[2,2] + m[3,3] + 4 \times 6 \times 2)$$

$$= m[2,2] + m[3,3] + 48 = 0 + 0 + 48$$

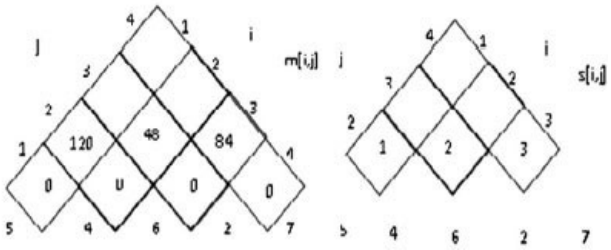
$$= 48, k = 2$$

$$m[3,4] = \min_{3 \le k \le 4} (m[3,3] + m[4,4] + 6 \times 2 \times 7)$$

$m[3,3]+m[4,4]+84$

$= 0+0+84$

$=84, k=3$



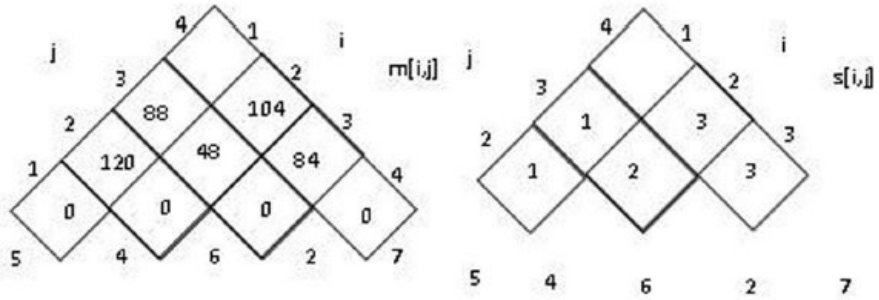
Step 4: Now we will compute value of m [1, 3]

$$m[1,3] = \min_{1 \le k \le 3} (m[1,k,] + m[k+1,3] + p_0 \times p_k \times p_3)$$

$$= \min \begin{cases} m[1,1] + m[2,3] + p_0 p_1 p_3 \\ m[1,2] + m[3,3] + p_0 p_2 p_3 \end{cases}$$

$$= \min \begin{cases} 0 + 48 + 5 \times 4 \times 2 = 88 \\ 120 + 0 + 5 \times 6 \times 2 = 160 \end{cases}$$

$$= 88, k = 1$$



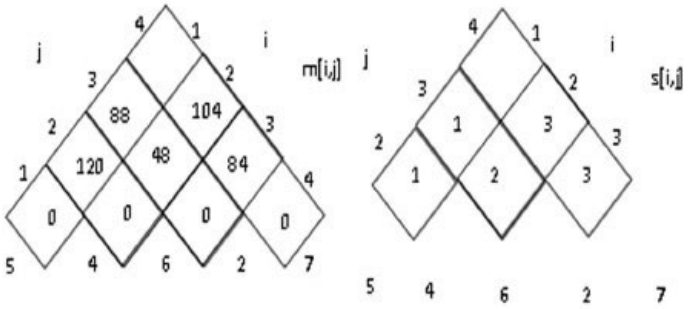
Step 5: Now we will compute value of m [2, 4]

$$m[2,4] = \min_{2 \le k \le 4} (m[2,k] + m[k+1,4] + p_1 \times p_k \times p_4)$$

$$= \min \begin{cases} m[2,2] + m[3,4] + p_1 p_2 p_4 \\ m[2,3] + m[4,4] + p_1 p_3 p_4 \end{cases}$$

$$= \min \begin{cases} 0 + 84 + 4 \times 6 \times 7 = 252 \\ 48 + 0 + 4 \times 2 \times 7 = 104 \end{cases}$$

$$= 104, k = 3$$



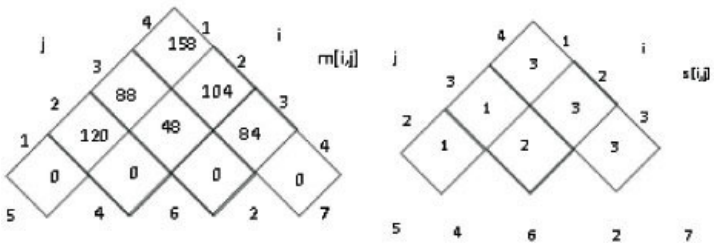
Step 6: Now we will compute value of m[1,4]

$$m[1,4] = \min_{1 \le k \le 4} (m[1,k] + m[k+1,4] + p_0 \times p_k \times p_4)$$

$$= \min \begin{cases} m[1,1] + m[2,4] + p_0 p_1 p_4 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 \\ m[1,3] + m[4,4] + p_0 p_3 p_4 \end{cases}$$

$$= \min \begin{cases} 0 + 104 + 5 \times 4 \times 7 = 104 + 140 = 244 \\ 120 + 84 + 5 \times 6 \times 7 = 204 + 210 = 414 \\ 88 + 0 + 5 \times 2 \times 7 = 88 + 70 = 158 \end{cases}$$

$$= 158, k = 3$$



Step 7: Now we compute places of parenthesis using PRINT OPTIMAL_SOLUTION (s,1,4)

And the resultant order of paranthesis will be ((A 1 (A 2 A 3)) A 4).

5.3 Longest Common Subsequence

A subsequence is a sequence of characters that can be derived from another sequence of character by deleting some elements from series without changing the order of the remaining elements. Longest common subsequence (LCS) of two sequences is a subsequence, with maximal length, which is common to both the sequences.

Suppose we have two strings X = and Y = then Y will be the subsequence of X iff there exist a strictly increasing sequence of *such that for all j=1,2, …,k we have x ij = y j . For example X= and Y= then for the given sequence longest common sequence (LCS) will be .*

Step 1: Characterizing a longest common subsequence: A brute-force approach for solving the LCS problem is to enumerate all possible subsequences of X and check the possible occurrence of each subsequence of Y, keeping track of the longest subsequence found. Each subsequence of string X of length m corresponds to a subset of the indices {1, 2, …, m} of X. It implies that there must be 2 m subsequences of X, so this approach requires exponential time, making it impractical for long sequences.

The LCS problem has a Optimal Substructure by using the prefix of two sequences.

Theorem 1: (Optimal Substructure of LCS)

Let two strings of length n and m are X = and Y = respectively and Z = be the LCS of X and Y of length k such that k ≤ n or k ≤ m.

If x n ==y m , then z k ==x n = y m and Z k-1 is an LCS of X n-1 and Y m-1 .

If x n ≠ y m , then z k ≠ x n implies that Z is an LCS of X n-1 and Y.

If x n ≠ y m , then z k ≠ y m implies that Z is an LCS of X and Y n-1 .

Proof (1) If zk≠ xn, then we could append xn = ym to Z to obtain a common subsequence of X and Y of length k + 1, contradicting the supposition that Z is a longest common subsequence of X and Y. Thus, we must have zk= xn = ym.

Now, the prefix Z k-1 is of length (k - 1), common subsequence of X n-1 and Y m-1 . Suppose for the contradiction there is a common subsequence W of X n-1 and Y m-1 with length greater than (k - 1). Then, appending x n = y m to W will produce a common subsequence of X and Y whose length is greater than k, which is a contradiction.

(2) If z k ≠ x n , then Z is a common subsequence of X n-1 and Y. If there were a common subsequence W of X n-1 and Y with length greater than k, then W would also be a common subsequence of X n and Y, contradicting the assumption that Z is an LCS of X and Y.

(3) The proof is symmetric to (2).

Step 2: A recursive approach: From the above theorem we came to know that to find LCS of X and Y, we may need to find the common LCS's of X and Y m-1 and of X n-1 and Y. But each of these sub problems can be further split into problem of finding the LCS of X n-1 and Y m-1 and so on.

The basic idea is to compute the longest common subsequence for each and every possible pair of prefixes. Let c [i; j] be the length of the longest common subsequence of X i and Y j . Here we are interested incalculating c[m; n], the LCS of the two entire strings. There are three steps for solving-

Basic : c [i,0]=c[j,0]=0 ⇒if either subsequence is zero then LCS will be empty.

Last character matches: Suppose X= and Y = < B, D, A>

Since here the last character 'A' matches then we can add it to the end of LCS and compare the sequence for X i-1 and Y j-1 .

Thus if x i =y j then c [i, j]= c [i -1, j-1]+1

Last character do not matches: suppose x i ≠y j then x i and y j can't be the part of LCS then in this case LCS of X i and Y j can be LCS of -

X i-1 and y j => c [i-1, j] or

X i and y j-1 => c [i, j-1]

Since we don't know which will be the case out of these two we may write-

If x i ≠y j then c[i, j] = max{c [i-1, j], c [i, j-1]}

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1]+1 & \text{if i, } j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if i, } j > 0 \text{and } x_i \neq y_i \end{cases}$$

Step 3: Computing the length of LCS: Procedure LCS-LENGTH takes two sequences X = and Y = as inputs. It stores the values ofc[i, j], which are computed in row wise manner of a table. (That is, the first row of c is filled in from left to right, then the second row, and so on.) It also maintains one more table b[i, j] for optimal solution, which points to the table entry corresponds to the optimal solution of sub problem, chosen at the time of computing c[i, j]. The procedure returns the b and c tables; c[m, n]contains the length of an LCS of X and Y.

Algorithm

```
1. LCS-LENGTH(X, Y)
2. n ← length[X]
3. m ← length[Y]
4. for i ← 1 to n
        a. do c[i, 0] ← 0
5. for j ← 0 to m
        a. do c[0, j] ← 0
6. for i ← 1 to n
        a. do for j ← 1 to m
        i. do if xi ==yj
                then c[i, j] ← c[i - 1, j - 1] + 1
                b [i, j] ← "↖";
        ii. else if c[i - 1, j] ≥ c[i, j - 1]
                then c[i, j] ← c[i - 1, j]
                b[i, j] ←"↑"
        iii. else if c[i, j] ← c[i, j - 1]
                then b[i, j] ← "←"
7. return c and b
```

Step 4: Constructing LCS: The table b constructed in LCS its length can be used to construct a LCS whenever we encounter a symbol ↖

in table its means that the x i = y j . In the reverse order the Print_LCS algorithm can be used to find LCS.

Algorithm

```
1.  PRINT_LCS(b, X, i, j)
2.  if i = 0 or j = 0
3.   then return
4.   if b[i, j] = " ↖"
     thenPRINT_LCS(b, X, i - 1, j - 1)
     print xi
5.   else if b[i, j] = "↑"
     then PRINT_LCS(b, X, i - 1, j)
6.   else PRINT_LCS(b, X, i, j - 1)
```

Analysis: The Time complexity of Procedure LCS-LENGTH () is O(n×m) and of PRINT_LCS() is also O(n×m)

Example: For the strings X= AND Y= find the longest common subsequence.

Step 1: Put all c[i,0] and c[0,]=0

Step 2: Now start evaluating from second row according to cases of c[i,j]

- For c[1,1] $x_1 = y_1$=B

⇒ c[1,1]=0+1=1

And b[1,1]= ↖

Same is the case for c[1,4] and c[1,6]

- Similarly, for c[1,2] $x_1 \neq y_2$

⇒ c[1,2]=max{c[1,1], c[0, 2]}=c[1,1] and b [1,2]=←

Same is the case for c[1,3] and c[1,5].

Step 3: Same rules will be followed in third row and the row will be

- For c[2,1] $x_2 \neq y_1$ ⇒ c[2, 1]=max{ c[1,1], c[2,0]}=c[1,1] and b[2,1]=↑
- For c[2,3], $x_2 \neq y_3$ ⇒ c[2, 3]= c[1, 2]+1=2 and b[2,3]=↖
- For c[2,4], $x_2 \neq y_4$ ⇒ c[2, 4]= max{c[1, 4], c[2, 3]}= c[2, 3] and b[2,4]=←

Same is the case for c[2,5] and c[2,6].

Step 4: Same steps will be followed for the remaining rows will be

Step 5: The element i the lowest right corner i.e. value of c[6, 6]=4 is the length of LCS.

To reconstruct the elements of an LCS, follow the b[i,j] arrows from the lower right hand corner; the path is shaded. Each " ↖

" on the path corresponds to an entry (highlighted for which x i = y j is a member of an LCS.

Step 6: Constructing LCS

1	Call Print_LCS(b, X, 6, 6) i, j ≠0 b [6, 6] =←, **Call Print_LCS(b, X, 6, 5)**				
2	Call Print_ LCS(b, X, 6, 5) i, j ≠0 2.1 b [6, 5] =↖,	2.2.Call Print_ LCS(b, X, 5, 4) i, j ≠0 • b [5, 4] = ↖	• **Print_LCS(b, X, 4 ,3)** i, j ≠0 a) b [4, 3]=↑ b) Call Print_LCS(b, X, 3 ,3)		
	2.2.Call Print_ LCS(b, X, 5, 4)	• **CCallPrint_ LCS(b, X, 4 ,3)** • **Print x_5 ='B'**	b) Print_LCS(b, X, 3 ,3) i, j ≠0 I. b [3, 3]=↑ II. call Print_LCS(b, X, 2 ,3)		
	2.3 Print x_6 ='A'		II. Print_LCS(b, X, 2 ,3) i, j ≠0 ✓ b [2, 3]= ↖ ✓ Call Print_ LCS(b, X, 1 ,2) ✓ Print x_2='C'	✓ **Print_LCS(b, X, 1 ,2)** i, j ≠0 ➢ b [1, 2]=↖ ➢ Call Print_ LCS(b, X, 0 ,1) ➢ Print x_1='B'	➢ **Print_LCS (b, X, 0 ,1)** **i= 0, j ≠0** **return i and j**

Output LCS is BCBA

The Longest Common Subsequence is Bcba

5.4 Optimal Binary Search Tree

If we are having a huge amount of data than to search it in lesser time we need to arrange it in an appropriate manner. Optimal binary search tree is a method in which we arrange the data in the form of binary search tree where their probability p i of usage and their contribution q i plays a major role.*

Suppose we have a set of sorted keys K= , for each value of ki we have probability p i that indicate the search for k i . Some searches may happen for the values, not present in K for this we have set of dummy keys D= , for each dummy key d i we have probability q i that indicate the contribution for d i . Each k i is an internal node and each d i is a leaf of binary search tree.

Every search is either successful (we are searching for k i) or unsuccessful (searching for dummy keys).

Step 1: The structure of an optimal binary search tree: To construct optimal binary search tree T we have to consider the sub-trees where each sub-tree contains all keys k 1 , k 2 , …,k n as an internal nodes and dummy keys d 0 , d 1 , d 2 ,… d n as leaf nodes. The left sub-tree will contain all keys from k 1 , k 2 , …, k r-1 with some dummy keys from d 0 , d 1 ,…, d r-2 and right sub tree will contains keys start from k r ,…..,k n having rest of the dummy keys from d r-1 to d n .

Step 2: A recursive solution: We pick our sub-problem for arranging it in form of an optimal binary search tree containing the keys k i ,…..,k j , where i≥1, j ≤n, and j≥i - 1. Let e[i, j] be the expected cost of searching an optimal binary search tree containing the keys k i ,…..,k j . Here, we wish to compute e [1, n], the expected cost of searching the nodes 1 to n of binary tree.

One possibility is, when j = i - 1. (It is when there are no actual keys) Then we have only the dummy key d i-1 . In such situation the expected search cost is e[i, i - 1] = q i-1 .

When j≥i, we need to select a root k r from the set of k i ,…..,k j and then make an optimal binary search tree with the help of keys k i ,…..,k r-1 as nodes of left sub-tree and with keys k r+1 ,…..,k j as the nodes of its right sub-tree.

Thus, if k r is the root node of an optimal sub-tree, containing keys k i ,…..,k j , we must have

e[i, j] = p r + (e[i, r - 1] + w(i, r - 1)) + (e[r + 1, j] + w(r + 1, j)).

where w (i, j) = w (i, r - 1) + p r + w(r + 1, j)

The e [i , j] values give the expected search costs in optimal binary search trees. To keep track of the structure of optimal binary search trees, we define root [i , j], for 1 ≤ i ≤ j ≤ n , to be the index r for which k r is the root of an optimal binary search tree containing keys k i ,…., k j .

Step 3: Computing the expected search cost of an optimal binary search tree: Here we stores the value of cost e[i,j] in a table, where 1≤i≤ n+1 and 0≤j≤n.

We also need to store value of root [i, j], containing keys k i ,…..,k j to generate an optimum cost tree where entries will be only for 1≤i≤n+1 and 0≤j≤n.

And one more table for w [i, j] where value of w [i, j] is required to compute e[i, j]. For the base case, we compute w [i, i - 1] = q i-1 for 1 ≤ i ≤ nand j≥i.

Algorithm

OPTIMAL-BST (p, q, n)

1. for i ← 1 to n + 1
   a. do e[i, i - 1] ← q_{i-1}
   b. w[i, i - 1] ← q_{i-1}
2. for l ← 1 to n
   a. do for i ← 1 to n - l + 1
      i. do j ← i + l – 1
      ii. e[i, j] ← ∞
      iii. w[i, j] ← w[i, j - 1] + p_j + q_j
      iv. for r ← i to j
1. do t ← e[i, r - 1] + e[r + 1, j] + w[i, j]
2. if t < e[i, j]
   I then e[i, j] ← t
   II root[i, j] ← r
3. return *e* and *root*

Step 4: To build binary tree from these roots: In the above algorithm, we have computed all the possible roots of OBST, now we will build the optimal binary tree from these calculated roots.

Algorithm

BUILD_OBST(i,j)

1. if i==j
2. then p ←0
3. else p->left← BUILD_OBST (i, root(i ,j)-1)
   p->key ←key(root(i ,j))
   p->right← BUILD_OBST (i+root (i , j),j)
4. return p

Analysis: The OPTIMAL-BST procedure takes Θ(n ³) time, just like MATRIX-CHAIN- ORDER. It is easy to see that the running time is O (n ³), since it's for loops are nested three deep and each loop index takes on at most n values.

Example:

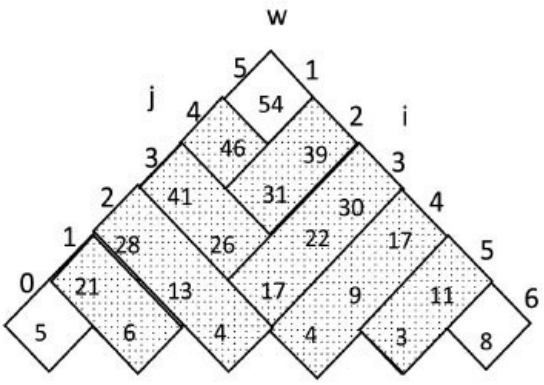
i	P_i	q_i
0		5
1	10	6
2	3	4
3	9	4
4	2	3
5	0	8

Step 1: Put for i ← 1 to n + 1

do e[i, i - 1], w[i, i - 1] ← q i-1

Step2: Compute w[i, j]

W [1,1] = w[1,0] + p1+q1= 5+10+6 =21
W [2,2] = w[2,1]+p2+q2=6+3+4=13
W[3,3] = 4+9+4= 17
W[4,4] = 4+2+3= 9
W[5,5] = 3+0+8=11
W[1,2] = 21+ 3+4= 28
W[1,3] = 28+9+4=41
W[1,4] = 41+2+3=46
W[1,5] = 46+0+8=54
W[2,3] = 13+ 9+4=26
W[2,4] = 26+2+3=31
W[2,5] = 31+0+8=39
W [3,4] = 17+2+3=22
W[3,5] = 22+0+8=30
W[4,5] = 9+0+8=17



Step 3: Compute e [i, j]

for r ← i to j

do t ← e[i, r - 1] + e[r + 1, j] + w[i, j]

root[i, j] ← r

e [1, 1]= e[1,0] +e[2,1]+w[1,1]

=5+6+21=32

e[2,2] = e[2,1]+e[3,2]+w[2,2]

=6+4+13=23

e[3,3]= 4+4+17= 25

e [4,4]= 4+3+9= 16

e[5,5] = 3+8+11=22

Computing next row- for r ← i to j

do t ← e[i, r - 1] + e[r + 1, j] + w[i, j]

if t < e[i, j]

then e[i, j] ← t

root[i, j] ← r

for computing e[1,2] r ← 1 to 2

e[1,2]= w[1,2]+ min (e[1,0]+e[2,2], e[1,1]+e[3,2])

= 28+ min(5+23,32+4)

= 28+ 28 = 56

⇒ r =1

Similarly for computing e[2,3] r ← 2 to 3

e[2,3]= w[2,3]+ min (e[2,1]+e[3,3], e[2,2]+e[4,3]) =26+ min(6+25,23+4)

=26+27=53

⇒ r =3

e[3,4] = w[3,4]+ min (e[3,2]+e[4,4],e[3,3]+e[5,4])

=22+ min (4+16, 25+3) =22+20=42

⇒ r = 3

e[4,5]=w[4,5]+min (e[4,3]+e[5,5],e[4,4]+e[6,5])

=17+ min(4+ 22, 16+8)

=17+24=41

⇒ r = 5

for computing e[1,3] r ← 1 to 3

e[1,3]= w[1,3]+ min(e[1,0]+e[2,3], e[1,1]+e [3,3], e[1,2]+e[4,3])

= 41+ min(5+53, 32+25, 56+4)

=41+57= 98

⇒ r =2

for computing e[2,4] r ← 2 to 4

e[2,4] = w[2,4] + min(e[2,1]+e[3,4],e[2,2]+e[4,4],e[2,3]+e[5,4])

= 31+ min(6+42, 23+16, 53+3)

= 31+ 39 = 70

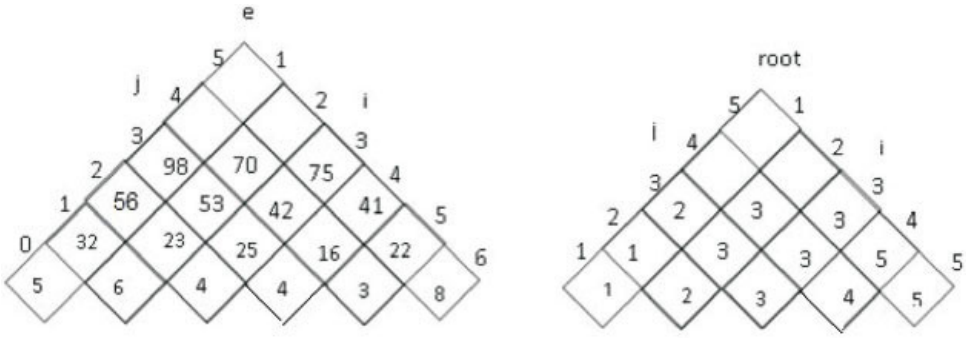
⇒ r = 3

Similarly for computing e[3,5] r ← 3 to 5

e[3, 5] = w[3,5] + min (e[3,2]+e[4,5], e[3,3]+e[5,5],e[3,4]+e[6,5])

= 30 + min (4+ 41, 25+22, 42+8) = 30 + 45=75

⇒ r = 3



for computing e[1,4] r ← 1 to 4

e [1, 4]= w[1,4] + min(e[1,0]+e[2,4], e[1,1]+e[3,4],e[1,2]+e[4,4]+e[1,3]+e[5,4])

= 46 + min(5+70,32+42, 56+ 16,98+ 3) =46+72= 118

⇒ r = 3

for computing e[2,5] r ← 2 to 5

e [2, 5] = w[2,5] + min (e[2,1]+ e[3,5], e[2, 2]+ e[4,5], e[2, 3]+ e[5,5], e[2, 4]+ e[6,5])

= 39 + min (6+ 75, 23+41, 53+22, 70+8) = 39 + 64 = 103

⇒ r = 3

Finally to compute e[1,5] r ← 1 to 5

e [1, 5] = w[1,5] + min(e[1,0]+e[2,5], e[1,1]+e[3,5],e[1,2]+e[4,5],e[1,3]+e[5,5],e[1,4]+e[6,5])

= 54+ min(5+103, 32+ 75,56+41,98+22, 118+ 8) = 54+ 97 = 151

⇒ r = 3

Step 4: build OBST

BUILD_OBST(1,5)

if i=j

then p ←0

else p->left← BUILD_OBST (i, root(i,j)-1)

p->key ←key(root(i,j))

p->right← BUILD_OBST (i+root (i, j),j)

return p

<table>
<tr><td>1</td><td colspan="3">BUILD_OBST(1,5)</td></tr>
<tr><td>2</td><td colspan="3">i≠j</td></tr>
<tr><td>3</td><td rowspan="3">p->left←BUILD_OBST (1, 3 -1)← BUILD_OBST (1,2)</td><td colspan="2">3 a) i≠j</td></tr>
<tr><td>b) p->left← BUILD_OBST (1,1-1)← BUILD_ OBST(1,0)</td><td>Block state as we don't have value of root(1,0)</td></tr>
<tr><td colspan="2">c) p->key ←key(root(1,2))←key(1)</td></tr>
<tr><td></td><td>d) p->right← BUILD_OBST (1+1,2)</td><td>I ==j
p←0 && p-> key ←key(root(2,2)) ←key(2)</td></tr>
<tr><td>4</td><td colspan="3">p->key ←key(root(1,5))←key(3)</td></tr>
<tr><td>5</td><td>p->right← BUILD_OBST (root (1,5),5)← BUILD_OBST (1+3,5)← BUILD_OBST (4,5)</td><td colspan="2">5a)) i≠j</td></tr>
<tr><td></td><td>b)) p->left← BUILD_OBST (4,5-1)</td><td>I ==j
p←0 && p-> key ←key(4)</td></tr>
<tr><td></td><td colspan="2">c) p->key ←key(5)</td></tr>
<tr><td></td><td>d) p->right← BUILD_ OBST(4+5,5)</td><td>Block state as we don't have value of root(9,5)</td></tr>
</table>

The Optimal Binary Tree with keys will be

Now we have to arrange dummy keys as well in this binary tree

Since w [2,2]= 13

And weight of active key (2) = 3

So here we can add two dummy keys d1 and d2.

Similarly w [4,4]=9

And weight of active key (4) = 2

So here we can add two dummy keys d2 and d4.

Similarly w [1, 1] =21

And weight of active key (1) = 10

So here we can add one dummy key d5.

Again w [5, 5] =11

And weight of key (5) = 0

So here we can add d0.

So the resultant optimal binary search tree will be

5.5 0/1 Knapsack Problem

The goal of Knapsack problem is to maximize the value (cost/profit) of knapsack having capacity of W units, from a list of items I 0 , I 1 , … I n-1 . Each item has two major attributes for consideration:

Value: Value may indicate the cost or the profit associate with a particular item, denoted by v i for item I i .

Weight: The weight of item and let this be w i for item I i .

Now, instead of taking a portion of certain weight of an item, you can worth take the item completely or not.

The naive method to find the solution of this problem is to iterate through all 2 n subsets of the n items and pick the subset with a legal weight that can maximize the value of the knapsack. Instead of iterating so many times, we can find a dynamic programming algorithm that will USUALLY do better than this brute force technique.

Our first step for dynamic strategy might be to characterize a sub-problem as follows:

Let Sk be the optimal subset of elements from {I 0 , I 1 ,…I k }. It may be possible that the optimal subset from the list of elements {I 0 , I 1 ,… I k+1 } may not correspond to the optimal subset of elements from the set {I 0 , I 1 ,…I k } in any regular pattern. Basically, the solution to the optimization problem for B k+1 might NOT contain the optimal solution from problem Bk.

Step 1 Optimal Substructure: To consider all subsets of items, there can be two possible cases for every item:

The item is either included in the optimal subset, (2) or not included in that set. Therefore, the maximum value that can be obtained from n items is optimum of the following two conditions:

Maximum value obtained by n-1 items and W weight (excluding nth item).

Value of n th item added to the maximum value obtained by n-1 items and W minus weight of the n th item (including n th item).

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

Step 2 Recursive Formula:

First case: w k > w. Item k can't be part of the solution, since if it was, the total weight would be greater than w, which is unacceptable.

Second case: w k ≤ w. Then the item k can be included, and we choose the case with greater value.

Recursive formula for sub problem is:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{If } w_i > w \\ \max(B[k-1,w], B[k-1,w-w_k+b_k]) & \text{otherwise} \end{cases}$$

Step 3 Computing the maximum value for the knapsack: Here we stores the value of B[i, w] in a for all given n items in the table. We will compute values of B[i, w] where 0 ≤i≤ n-1 and 0≤w≤n.

Algorithm

1. for $w \leftarrow 0$ to W
2. do $B[0, w] = 0$
3. for $i \leftarrow 1$ to n
4. do $B[i, 0] = 0$
   a. for $w \leftarrow 1$ to W
   b. do if $w_i \le w$ // item I can be a part of the solution
   c. then if $b_i + B[i\text{-}1, w\text{-}w_i] > B[i\text{-}1, w]$
      i. then $B[i, w] = b_i + B[i\text{-}1, w\text{-}w_i]$
      ii. else $B[i, w] = B[i\text{-}1, w]$
   d. else $B[i, w] = B[i\text{-}1, w]$ // $w_i > w$

Step 4 Finding the Optimal Subset of Items For The Given Knapsack:

Algorithm

B [n, w] will be the maximum value that we can put in knapsack.

Let i = n and k = W

whilei, k > 0

A. If B [i, k] ≠ B[i-1, k]

B. then mark the i th item

i = i-1, k = k - w i

else i = i-1 // assume that i th item is not included in knapsack

Analysis: The loop for i will execute = n times

And the inner w loop will execute = n × W times

Hence the complexity will be O(n × W)

Example: Find the subset of items for the knapsack of capacity 5 and the items we have are as follows:

S. No.	Weight w_i	Value b_i
1	2	3
2	3	4
3	4	5
4	5	7

Step 1: Make all B[i,0] and B [0,w] equals to zero

Step 2: i=1 and w=1

w 1 =2<1

if b i +B[0,1,-2]>B[0,-1]>B[0,1]

else B[1,1]=B[0,1]=0

Step 3: Similarly for i=1 and w=2

w 2 =2<2

if b 1 +B[0, 2-2]>B{0,2]=>3+0>0

⇒ B[1,2]=3

Similarly, B[1,3], B[1,4] and B[1,5]=3

Step 4: Now, we will computer for i=2 and w=1

w 2 =3<1

if b 1 +B[1,1-2]>B[1,1]=>3+B[1,-1>0

else B[2,1]=B[1,1]=0

Step 5: Now, we will computer for i=2 and w=2

w 2 =3>2

if b 2 +B[1,1-2]>B[1,1]=>3+B[1, -1]>0

else B[2,2]=B[1,2]=3

Step 6: Now, we will computer for i=2 and w=3

w 2 =3>=3

if b 2 +B[1,0]>B[1,3]=> 4+0>3

then B[2,3]=4+0=4

Step 7: Now, we will computer for i=2 and w=4

w 2 =3<=4

if b 2 +B[1,1]>B[1,4]=>4+0=4

B[2,4]=4

Step 8: Now, we will computer for i=2 and w=5

w 2 =3<=5

if b 2 +B[1,2]>B[1,5]=>4+3>3

B[2,5]=7

Step 9: Similarly, we will compute for i=3 and w=1 to 3 We will get B [3,1]=0, B[3,2]=3, B[3,3]=4

Step 10: Now, we will compute for i=3 and w=4

w 3 =4<=4

if b 3 +B[2,0]>B[2,4]=>5+0>4

B[3,4]=5

And for w=5

w 3 =4<=5

if b 3 +B[2,1]>B[2,5]=>5+0>7(not true)

else B[3,5]=B[2,5]=7

Step 11: Now we will compute for i=4 and w=1 to 5

The resultant will be

Step 12: Start from the lowest right corner i.e. from B[4, 5] to pick items for knapsack.

Since B[4, 5] == B[3,5], we will go to next item without selecting any of these.

Now i = 3

Check B[3,5] = = B[2,5] we will go to next item without selecting none of these.

Now i = 2

Since B[2,5]≠ B[1,5]

Item 2 is selected for knapsack.

Again for i = 1

Since B[1,5]≠ B[0,5]

Item 1 is selected for knapsack.

Hence, the optimal solution for the given problem has item {1, 2} in the knapsack.

5.6 Travelling Salesman Problem

A traveling salesman is going for a long sales tour to visit multiple locations in a single day. Starting at his hometown, having suitcase in hand, he will conduct a journey in such a manner that each of his targeted cities is visited exactly once before he returns to the home, such that we will travel an optimal path. For finding the optimum path, he has pair-wise distances between cities. All he has to find is what will be the best order to visit them, to minimize the overall distance covered? Figure shows an example involving four cities, city 1 is the starting point and the distance between cities.

A TSP tour for the above graph is 1-2-4-3-1. The overall cost of this tour will be 10+25+30+15 which is 80.

The traveling: Salesman problem, is closely related to the Hamiltonian-cycle problem where a salesman must visit all n cities. Modeling the problem as a complete graph with n vertices, we can say that the salesman wishes to make a tour, or Hamiltonian cycle, visiting each city exactly once and finishing at the city from where he starts the journey. There is an integer cost c(i, j) associated to travel from city into city j, and the salesman willing to make the tour in such a way so that overall cost of tour will be minimized, where the total cost is given the sum of the individual costs along the edges of the tour.

The problem is a famous NP hard problem. There is no polynomial time know for the solution of this problem.

Optimal substructure: In this case the most obvious partial solution is the initial portion of a tour. Suppose we started at city 1, have visited a few cities in between, and are now in city j. What information do we need in order to extend this partial tour? We certainly need to know j (the neighboring cities of j and cost associated with them), since this will determine which city is most convenient to visit next. And we also need to know all the cities have been visited so far, so that we don't repeat any of them.

A recursive approach: Denote the cities by 1,…,n, the salesman's hometown being 1, and let c = (c ij) be the matrix of intercity distances. The goal is to design a tour that starts and ends at city 1, includes all other cities exactly once, and has minimum total length of tour.

For a subset of cities $S \subseteq \{1,2,\ldots,n\}$ that includes 1, and $j \in S$, let g(S,j) be the length of the shortest path of the tour such that visiting each node in S exactly once, starting from city 1 and ending at city j.

When $|S| > 1$, we define $g(S, 1) = \infty$. Since the path cannot start and end at the same city 1 at a same time.

Now, let's express g(S,j) in terms of smaller sub-problems. We need to start at node 1 and end the route at some other node j; what node should we pick to move from 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, g(S - {j}, i), plus the length of the final edge,cij. We must choose the node I such that: $g(S,j)=\min i \in S:i \neq j\ g(S-\{j\},i)+c\ ij$

Computing the optimal cost: The next step of the dynamic programming paradigm is to construct the value of an optimal solution in a bottom-up fashion. In our problem the only tricky part is arranging the order to compute the values (so that it is readily available when we need it). It is easy to see that if the tour is optimal then the path from k to 1 must be shortest then k to 1 path going through all vertices in V - {1, k }. Hence, the principle of optimality holds. Let g(i,S) be the length of a shortest path starting at vertex i, going through all vertices in S and terminating at vertex 1. g (1, V - {1 }) is the length of an optimal tour. From the principal of optimality it follows that: $g(l, V - \{1\}) = \min \{c\ 1k + g(k, V- \{l,k\})\}$

Generalizing this equation we obtain (for $i \varepsilon S$) $g(i, S) = \min \{c\ ii + g(j, S - \{j\})\}$

Algorithm

$g\{1\},1) = 0$

for s = 2 to n:

for all subsets $S \leftarrow \{1,2,\ldots,n\}$ of size s and containing 1:

i $g(S,1) = \infty$

ii for all $j \varepsilon S, j \neq 1$:

$g(S, j) = \min\{g(S-\{j\},i)+c\ ij : i \varepsilon S, i \neq j\}$

iii return min j $g(\{1,\ldots,n\},j)+c\ j1$

Example :

$$C = \begin{pmatrix} 0 & 2 & 10 \\ 1 & 0 & 4 \\ 15 & 7 & 8 \\ 6 & 3 & 0 \end{pmatrix}$$

Distance matrix is given by c here.

$g(2, \theta) = C_{21} = 1$
$g(3, \theta) = C_{33} = 15$
$g(3, \theta) = c_{41} = 6$
K = 1, consider sets of 1 element:

Set {2}: $g(3,\{2\} = C32 + g(2,q) = C32 + C21 = 7 + 1 = 8$ $p(3,\{2\}) = 2$
$g4(4,\{3\} = C42 + g(2,q) = C42 + C21 = 3 + 1 = 4$ $p(4,2\{2\}) = 2$

Set {3}: $g(2,\{3\}) = C23 + g(3,q) = C23 + C31 = 6 + 15 = 21$ $p(2,\{3\}) = 3$
$g(4,\{3\}) = C43 + g(3,q) = C43 + C31 = 12 + 15 = 27$ $p(4,\{3\}) = 3$

Set {4} $g(2,\{4\}) = C24 + g(4,q) = C24 + C41 = 4 + 6 = 10$ $p(2,\{4\}) = 4$
$g(3,\{4\}) = C34 + g(4,\theta) = C_{34} + C41 = 8 + 6 = 14$ $p(3,\{4\}) = 4$

k = 2, consider sets of 2 elements:

Set{2,3}: $g(4,\{2,3\}) = \min \{42 + g(3,\{2\}), + g(3,\{2\})\} = \min (3 + 21, 12+8\} = \min \{24, 20\} = 20$
$p(4,\{2,3\}) = 3$

Set{2,4}: $g(3,(2,4\}) = \min \{C32 + g(2,\{4\}), C34 + g(4,\{2\})\} = \min \{7 + 10, 8 + 4\} = \min (17, 12) = 12$
$P(3,\{2,4\}) = 4$

Set {3,4}: $g(2,\{3,4\}) = \min \{C32 + g(3,(4), C_{24} + g(4,\{3\})\} = \min \{6 + 14, 4+27\} = \min \{20, 31\} = 20$
$p(2,\{3,4\}) = 3$

Length of an optimal tour:

$f = g(1, \{2, 3, 4\}) = \min \{C_{12} + g(2,\{3,\{2,4), C_{13} + g(4, 3\ \{2, 4\}), C_{14} + g(4,\{2, 3\})\ \}$
$= \min \{2+20, 9 + 12, 10 + 20\} = \min \{22, 21, 30\} = 21$
Successor of node 1 : $p(1,\{2, 3, 4\}) = 3$
Successor of node 3 : $p(3, \{2,4\}) = 4$
Successor of note 4 : $p(4, \{2\}) = 2$
Optimal TSP tour: 1 >> 3 >> 4 >> 2 >> 1

Computing Optimal Solution : To know the path, first note that the value of 21 is a minimum value obtained from 9+12. Means the shortest path is c 13 +g(3,{2,4}) where found that the initial part of the path is 1-3. Then find out the component of g(3,{2,4}) which is as the minimum value of c 34 +g(4,{2}) Therefore, it is known that the edge 3-2 is part of the shortest path.

In the same way, we trace the g(4,{2}) which is found that c 42 +g(2,∅). The final step is to trace forming of g(2,∅) which found is found to be h c 41 . (Thus edge of 4-1 is also part of the shortest path). By assembling the edges that has been defined as (edge 1-3, 3-2, 2-4, 4-1). This indicates that the shortest path is 1-3-2-4-1 with the tour of minimum distance of 21 units.

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

REVIEW EXERCISE

What is dynamic programming?

Explain 0/1 knapsack problem with the help of an example?

What is the difference between greedy and dynamic approach?

Find the longest common subsequence between the strings ABAZDC and BABDCG.

Find the optimal solution using knapsack for-

	A	B	C	D	E	F	G
VALUE	7	9	5	12	14	6	12
TIME	3	4	2	6	7	3	5

Solve and find the solution for matrix chain multiplication n=4. A1 is 3 x 5, A2 is 5 x 7, A3 is 7 x 3, and A4 is 3 x 4.

Calculate the OBST for p1=1/10, p2=2/10, p3=3/10, p4=1/10 and q0=0, q1=1/10, q2=1/20, q3=1/20, q4=1/10.

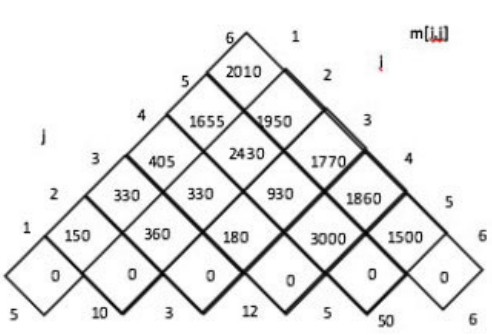
For the order p=<5, 10, 3, 12, 5, 50, 6> find the optimum solution using matrix chain multiplication.

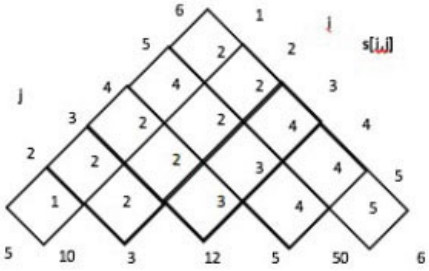
Suppose we want to make change for n cents, using the least number of coins of denominations 1; 10, and 25 cents. Describe an O(n) dynamic Suppose we want to make change for n cents, using the least number of programming.)

Given a "m x n" matrix, count number of paths to reach bottom right from top left with maximum k turns allowed.

Solution of Review Excercise

Ans 8





Resultant matrix chain will be (A1 A2) ((A3 A4) A5 A6).

Ans. 9 Below is a dynamic programming solution for this problem to illustrate how it can be used. There is a very straightforward O(1) time solution. It can be shown that if n >= 50 then any solution will include a set of coins that adds to exactly 50 cents. Hence it can be shown that an optimal solution uses 2 _ bn=50c quarters along with an optimal solution for making n=50 bn=50c cents which can be looked up in a table of size 50.

Here's the dynamic programming solution for this problem. (It does not use the fact that an optimal solution can be proven to use 2 _ bn=50c quarters and hence is not as ecient.) The general subproblem will be to make change for i cents for 1 _ i _ n. Let c[i] denote the fewest coins needed to make i cents. Then we can dene c[i] recursively by:

$$c[i] = \begin{cases} \text{use i pennies} & \text{if } 1 \le i < 9 \\ c[i..10] + 1 & \text{if } 10 \le i < 24 \\ c[i..10] + 1 & \text{if } 10 \le i < 24 \end{cases}$$

Note that c[n] is the optimal number of coins needed for the original problem.

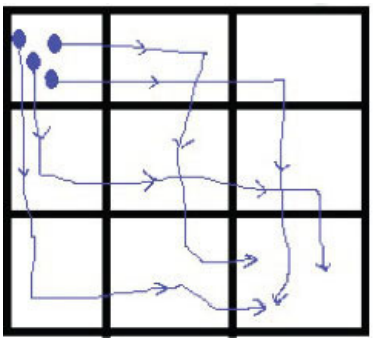
Clearly when i < 10 the optimal solution can use only pennies (since that is the only coin available). Otherwise, the optimal solution either uses a dime or a quarter and both of these are considered. Finally, the optimal substructure property clearly holds since the _nal solution just adds one coin to the subproblem solution. There are n subproblems each of which takes O(1) time to solve and hence the overall time complexity is O(n).

Ans. 10 What is a turn? A movement is considered turn, if we were moving along row and now move along column. OR we were moving along column and now move along row. There are two possible scenarios when a turn can occur at point (i, j):

Turns Right: (i-1, j) -> (i, j) -> (i, j+1)
Down Right
Turns Down: (i, j-1) -> (i, j) -> (i+1, j)
Right Down Examples:

Input: m = 3, n = 3, k = 2
Output: 4
See below diagram for four paths with maximum 2 turns.

Input: m = 3, n = 3, k = 1
Output: 2



Input: m = 3
n = 3
k = 2

Output: 4
Four paths are highlighted on left

MULTIPLE CHOICE QUESTIONS & ANSWERS

An algorithm to find the length of the longest monotonically increasing sequence of numbers in an array A[n] is as follows:

Initialize L n-1 =1

For all I such that 0≤i≤ n-2

Let Li denote the length of the longest monotonically increasing sequence starting at index i in the array.

Which of the following statements is TRUE for this algorithm?

The algorithm uses dynamic programming paradigm

The algorithm uses branch and bound paradigm and has a linear time complexity.

The algorithm uses branch and bound paradigm and has a non-linear polynomial time complexity.

The algorithm uses divide and conquer paradigm.

Ans. (a) The algorithm uses dynamic programming paradigm

There are two strings A = "qpqrr" and B = "pqprqrp". Let x be the length of the longest common subsequence between A and B and y be the occurrences of such longest common subsequences between A and B. Then value of x + 10y = ____.

a 33

b 23

c 43

d 34 [GATE 2014]

Ans. (d) 34

Explanation: There are 3 occurrences of LCS of length 4 "qprr", "pqrr" and qpqr

The basic property of Dynamic programming approach is

(a) It provides optimal solution

(b) The solution has optimal substructure

(c) The given problem can be reduced to the 3-SAT problem

(d) It's faster than Greedy approach.

Ans. (b) The solution has optimal substructure

Kadane's algorithm is used to find:

(a) Maximum sum of subsequence in an array

(b) Maximum sum of contiguous sub-array of an array

(c) Maximum product of contiguous subsequence in an array

(d) Maximum product of sub-array in an array

Ans. (b) Maximum sum of contiguous sub-array of an array

Explanation: Kadane's algorithm is used to find the maximum sum of contiguous sub-array of an array. It has O(n) time complexity.

Let A1, A2, A3, and A4 be four matrices of order 10 x 5, 5 x 20, 20 x 10, and 10 x 5, respectively. The minimum number of scalar multiplications required to find the product of these matrices using the naive matrix multiplication method is

(a) 1500

(b) 2000

(c) 500

(d) 100 [GATE 2016]

Ans. (a) 1500

Explanation: If we multiply two matrices A and B of order l x m and m x n respectively, then the number of scalar multiplications in the multiplication of A and B will be lxmxn.

Then, the number of scalar multiplications required in the following sequence of matrices will be:

A1 ((A2A3) A4) = (5 x 20 x 10) + (5 x 10 x 5) + (10 x 5 x 5) = 1000 + 250 + 250 = 1500.

There are two sequences X[m] and Y[n] of lengths m and n respectively, with indexes starting from 0. To find the length of the longest common sub- sequence(LCS) of X[m] and Y[n] as l(m,n), we have an incomplete recursive definition for the function l(i,j) as given below:

$$l(i,j) = \begin{cases} 0, & if\ i == 0\ or\ j == 0 \\ \text{expr1} & if\ i,j > 0\ and\ X[i+1] == Y[j-1] \\ \text{expr2} & if\ i,j > 0\ and\ X[i-1]! = Y[j-1] \end{cases}$$

Which statement is true:

(a) expr1 ≡ l(i-1, j) + 1

(b) expr1 ≡ l(i, j-1)

(c) expr2 ≡ max(l(i-1, j), l(i, j-1))

(d) expr2 ≡ max(l(i-1,j-1),l(i,j)) [GATE 2009]

Ans. (c)

Explanation: In Longest common subsequence problem, there are two cases for X[0..i] and Y[0..j]

The last characters of two strings match.

The length of LCS is length of LCS of X[0..i-1] and Y[0..j-1]

The last characters don't match.

The length of LCS is max of following two LCS values

a) LCS of X[0..i-1] and Y[0..j]

b) LCS of X[0..i] and Y[0..j-1]

C HAPTER -6

Graph Theory

In this chapter student will understand:

What is tree and graph? What are the different types of are of tree, how we represent them, traverse them? How the various shortest paths finding techniques work, which strategy is used to optimized them?

6.1 Binary Tree

A binary tree contains nodes, where each node contains a left reference, a right reference, and a data element. The node at the top is called the root of the tree.

Each node is connected to directed nodes to each other. A node (parent node) can be con- nected to an arbitrary number of nodes. The nodes having no child are called leaf nodes. Nodes which are not leaf nodes are called internal nodes.

Full binary tree: In this each node has exactly zero or two children.

Complete binary tree: A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.

A complete binar y: It provides a best ratio between number of nodes and height h.

h =1+2+4+…… +2 h -1 +2 h =2 h +1 -1

⇒ h =O(log n)

6.2 Binary Search Tree

The concept behind binary search tree, is to provide a data structure that stores data to perform sorting and searching operations. The structure of binary tree is as follows:

Each node contains a key.

The key value of left node contains data element whose value is less than that of parent node. L

The key value of right node contains data element whose value is greater than that of parent node. P

No duplicate keys are allowed.

6.3 Huffman Code

This is the example of application of binary trees with minimal weighted external path length to obtain an optimal set of codes for messages M 1, …., Mn+ 1. Each code is a binary string which is used for transmission of the corresponding message. At the receiving end the code will be

decoded by using a decode tree. A decode tree is a binary tree in which external nodes represent messages. The binary bits in the code word for a message is used to determine the branching needed at each level of the decode tree to reach the correct external node.

Huffman's greedy algorithm creates a table to record the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.

There are mainly two major parts in Huffman Coding

1) Build a Huffman Tree from input characters.

2) Traverse the Huffman Tree and assign codes to characters.

Algorithm

In the pseudo code that follows, we assume that C is a set of n characters and that each character c ε C is an object with a predefined frequency f [c]. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of |C| leaves and performs a sequence of |C| - 1 "merging" operations to create the final tree. A min-priority queue Q, keyed on f, is used to identify the two least-frequent objects to merge together. The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of these two objects that were merged.

HUFFMAN(C)

1 n ← |C|

2 Q ← C

3 for i→1 to n - 1

4 do allocate a new node z

5 left[z] ← x ← EXTRACT-MIN (Q)

6 right[z] ← y ← EXTRACT-MIN (Q)

7 f [z] ← f [x] + f [y]

8 INSERT(Q, z)

9 return EXTRACT-MIN(Q) //Return the root of the tree.

Example: Let a string has 6 letters in an alphabet means the initial size of queue is 6.

Character	Frequency
A	5
B	9
C	12
D	13
E	16
F	45

Note: Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters; lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, are assigned uniquely to all characters to avoid ambiguity.

For example there will be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be "cccd" or "ccb" or "acd" or "ab".

6.4 Graph

Graph is a data structure that is used to model a set of problems that are different in nature but related to each other in some manner. While tree represents a hierarchy of elements that are related to some elements.

In graph each vertex is denoted by V and edge by E ⇒ G ε(V , E).

The graph can be directed or undirected or may be weighted and unweighted one.

6.4.1 Representation of graphs

Following is the different ways to represent a graph:

6.4.1.1 Adjacency List : Directed graphs are commonly represented as an adjacency list, which comprises an array or list of vertices, where each vertex v i stores a list of all the vertices v j for which there is an edge (v i ,v j) εE

For the above graph adjacency list will be

6.4.1.2 Adjacency Matrix: Its a two-dimensional array, where value of Ai j is non- zero when there is an edge (v i , v j) ε E. Many graphs are sparse i.e. most of the possible edges between pairs of vertices do not exist, i.e. m << n ² . In such cases the adjacency list is generally preferable to the adjacency matrix representation.

	1	2	3	4	5	6
1	0	1	0	0	1	1
2	1	0	0	1	1	1
3	1	0	0	0	1	0
4	0	1	0	0	0	1
5	1	1	1	0	0	0
6	0	0	0	1	0	1

6.4.1.3 Weighted graphs: Most of graph, represents cost or distance on their edges.

6.4.1.4 Traversal of graph: The basic operation of graph is traversing, where we starts from a node and find all the nodes connect to that node. There are two methods two traverse a graph:

Breadth First Search: Given a graph G = (V, E) and a distinguished source vertex s, breadth-first search, explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a "breadth-first tree" with root s that contains all reachable vertices.

For any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the smallest number of edges.

Algorithm

```
1. BFS(G,s)
2. for each u εV - {s}                    // for loop
3. color[u]←white
a. d[u]←∞
4. π [u]←Φ
5. color[s]←gray
6. d[s]←0
7. π [s]←Φ
8. Q←Φ  // Initialize queue is empty
9. Enqueue(Q,s)                           /* Insert start vertex s in Queue Q */
10. while Q ≠Φ                            // while loop
    a.  u ← Dequeue[Q]                    /* Remove an element from Queue Q*/
    b.  for each v ∈ Adj[u]               // for loop
    c.if (color[v] == white)              /*if v is unvisted*/
        i.  color[v] ← gray;              /* v is visted */
        ii. d[v] ← d[u] + 1;              /*Set distance of v to no. of edges from s to u*/
        iii. π [v] = u;                   /*Set parent of v*/
        iv. Enqueue(Q,v);                 /*Insert v in Queue Q*/
  d. color[u] = black; /*finally visted or explored vertex u*/
```

Algorithm to print the resultant tree

PRINT-PATH(G, s, v)

if v = s

then print s

else if $\varpi[v] = \text{NIL}$

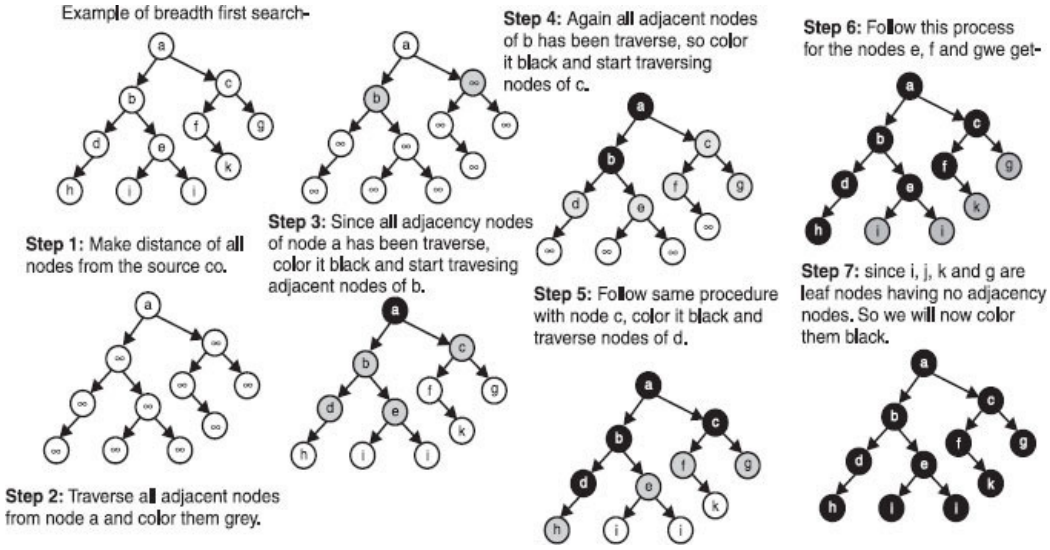
then print "no path from" s "to" v "exists"

else PRINT-PATH(G, s, $\varpi[v]$)

print v

Analysis: After initialization, no vertex is ever whitened, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once.

The operations of enqueuing and dequeuing take O(1) time, so the total time devoted to queue operations is O(V). Because the adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once. The sum of the lengths of all the adjacency lists is (E), the total time spent in scanning adjacency lists is O(E). The overhead for initialization is O(V), and thus the total running time of BFS is O(V + E).



Depth First Search: DFS in not so useful in finding shortest path. It is used to perform a traversal of a general graph and the idea of DFS is to make a path as long as possible, and then go back (backtrack) to add branches also as long as possible.

Broadly it is divided into 3 steps:

Take a vertex that is not visited yet and mark it visited

Go to its first adjacent non-visited (successor) vertex and mark it visited

If all the adjacent vertices (successors) of the considered vertex are already visited or it doesn't have any more adjacent vertex (successor) - go back to its parent vertex

Algorithm

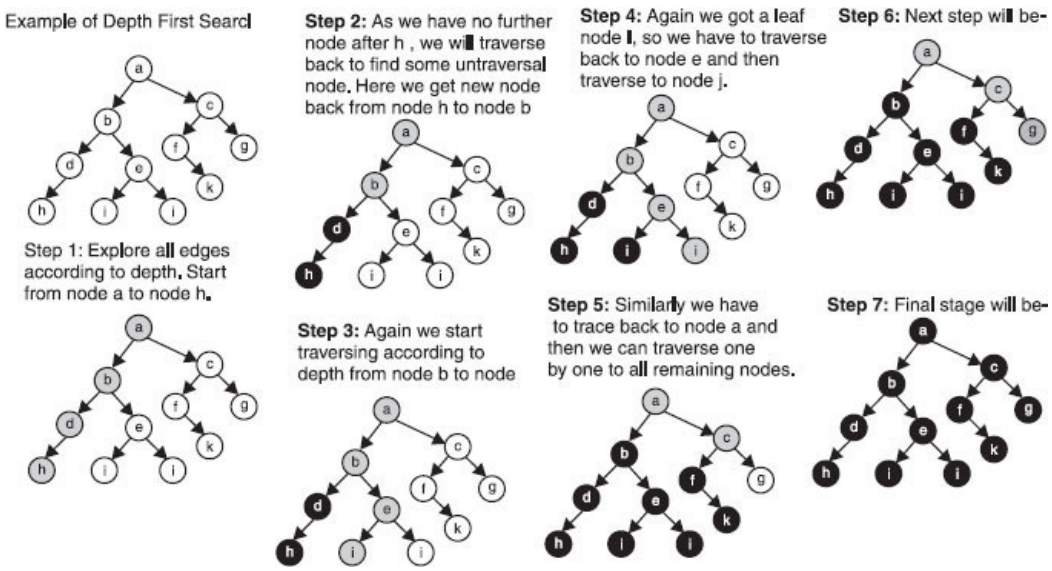
DFS(G)

1. **For** each vertex $u \in V[G]$
2. **Do** $color[u] \leftarrow$ WHITE
3. $\pi[u] \leftarrow$ NIL
4. $time \leftarrow 0$
5. **for** each vertex $u \in V[G]$
   i. **do if** $color[u]$ = WHITE
   ii. **then** DFS-VISIT(u)

DFS-VISIT(u)

1. $color[u] \leftarrow$GRAY //White vertex u has just been discovered.
2. $time \leftarrow time + 1$
3. $d[u] \leftarrow ++time$
4. **for** each $v \in Adj[u]$ //Explore edge(u, v).
5. **do if** $color[v]$ = WHITE
6. **then** $\pi[v] \leftarrow u$
7. DFS-VISIT(v)
8. $color[u] \leftarrow$ BLACK //Blacken u; it is finished.
9. $f[u] \leftarrow time + 1$

Analysis: The total cost of executing lines 4-7 of DFS-VISIT is Θ(E). The running time of DFS is therefore Θ(V + E).



Note: Breadth First Search algorithm works on both directed and undirected graphs. Prim's minimal spanning tree and Dijkastra's single source shortest path algorithm both use breadth first search concept.

6.4.15 Connected graph: In an undirected graph , a connected component is the set of nodes that are reachable by traversal from some node. The connected components of an undirected graph have the property that all nodes in the component are reachable from all other nodes in the component.

Whereas, In a directed graph , however, reachable usually means by a path in which all edges go in the positive direction, i.e. from source to destination. In directed graphs, a vertex v may be reachable from u but not vice-versa. In the above example of BFS, the set of nodes reachable from a are b,c… k. but we can't reach a from node k.

The strongly connected components in a directed graph are defined in terms of the set of nodes that are mutually accessible from one another. In other words, the strongly connected component of a node u is the set of all nodes v such that v is reachable from u by a directed path and u is reachable from v by a directed path. Equivalently, u and v lie on a directed cycle.

6.4.1.6 Topological Order (Topological Sorting): In a directed acyclic graph,the nodes can be ordered such that each node in the ordering comes before all the other nodes to which it has outbound edges. This is called a topological sort of the graph.

A topological sort of a dag G = (V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. (If the graph is not acyclic, then no linear ordering is possible.) A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.

There is not a unique topological order for a given directed acyclic graph. If there are cycles in the graph, there is no topological ordering. Topological orderings have many uses for problems ranging from job scheduling to determining the order in which to compute quantities that depend on one another (e.g., spreadsheets).

Algorithm

TOPOLOGICAL-SORT(G)

call DFS(G) to compute finishing times f [v] for each vertex v

as each vertex is finished, insert it onto the front of a linked list

return the linked list of vertices

Analysis: The running time of this method is O (n ²), whereas the asymptotically fastest methods are O (V+E).

6.5 Minimum Spanning Tree

Suppose we have an undirected graph G =(V,E) and for each edge (u, v)e E we have some weight w associated with it.

A sub graph that contains all vertices and acyclic subset of edges T such that weight of it given by $w(T) = \sum_{(u,v)\varepsilon T} w(u,v)$

Since T is acyclic and connects all vertices hence it must form a tree called spanning tree and represent by S = (V , T): T ⊆ E and the problem of determining spanning tree T is called minimum spanning tree problem.

Example: If a delivery boy has to deliver objects at various destinations 1,2,3,4 and 5 and these destinations are fully connected with each other then, to reduce his time and distance of travel, he decides his route on the basis of minimum distance covered by him by comparing distance between them.

Figure: Undirected weighted Graph

Figure: Spanning Tree

6.5.1 Generic Approach for Minimum Spanning Tree

We have a connected, undirected graph G = (V, E) with a weight function w (specifying the distance between nodes), and we wish to find a minimum spanning tree for G. This greedy strategy is captured by the following "generic" algorithm, which grows the minimum spanning tree by adding one edge at a time. The algorithm manages a set of edges A, maintaining the following loop invariant:

Prior to each iteration, A is a subset of some minimum spanning tree.

At each step, we determine an edge (u, v) that can be added to A without violating this invariant, in the sense that A□{(u, v)} is also a subset of a minimum spanning tree. We can call such an edge a safe edge for A, since it can be safely added to A while maintaining the invariant.

Algorithm

GENERIC-MST(G, w)

A ← ∅

while A does not form a spanning tree

do find an edge (u , v) that is safe to add in A

(a) A ← A ∪{(u , v)}

return A

We use the loop invariant as follows:

Initialization : After line 1, the set A trivially satisfies the loop invariant.

Maintenance : The loop in lines 2-4 maintains the invariant by adding only safe edges.

Termination : All edges added to A are in a minimum spanning tree, and so the set A is returned in line must be a minimum spanning tree.

There are two algorithms for describing minimum spanning tree:

Kruskal's algorithm

Prim's algorithm

6.5.2 Kruskal's Algorithm

It builds the MST in the form of forest. Initially, each vertex is a tree in its own in the forest. Then, algorithm considers each edge in turn, in order of increasing weight. The safe edge that will be connected to the forest is always a least weighted edge in graph that connects two distinct components without creating a cycle i.e. If an edge (u , v) connects two different trees, then (u , v) is added to the set of edges of the MST, and two trees connected by an edge (u , v) are merged into a single tree on the other hand, if an edge (u , v) connects two vertices in the same tree, then edge (u , v) is discarded.

A little more formally, given a connected, undirected, weighted graph with a function w : E → R.

Starts with each vertex being its own component.

Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them).

Scans the set of edges in monotonically increasing order by weight.

Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

Algorithm

Start with an empty set A, and select at every stage the shortest edge that has not been chosen or rejected, regardless of where this edge is situated in the graph.

KRUSKAL(V, E, w)

$A \leftarrow \{\}$ ▶

Set A will ultimately contains the edges of the MST

for each vertex v e V | G |

do MAKE-SET(v)

sort the edges E into non-decreasing order by weight w

for each (u , v) taken from the sorted list

do if FIND-SET(u) ≠ FIND-SET(v)

(a) then A ← A ← {(u , v)}

return A

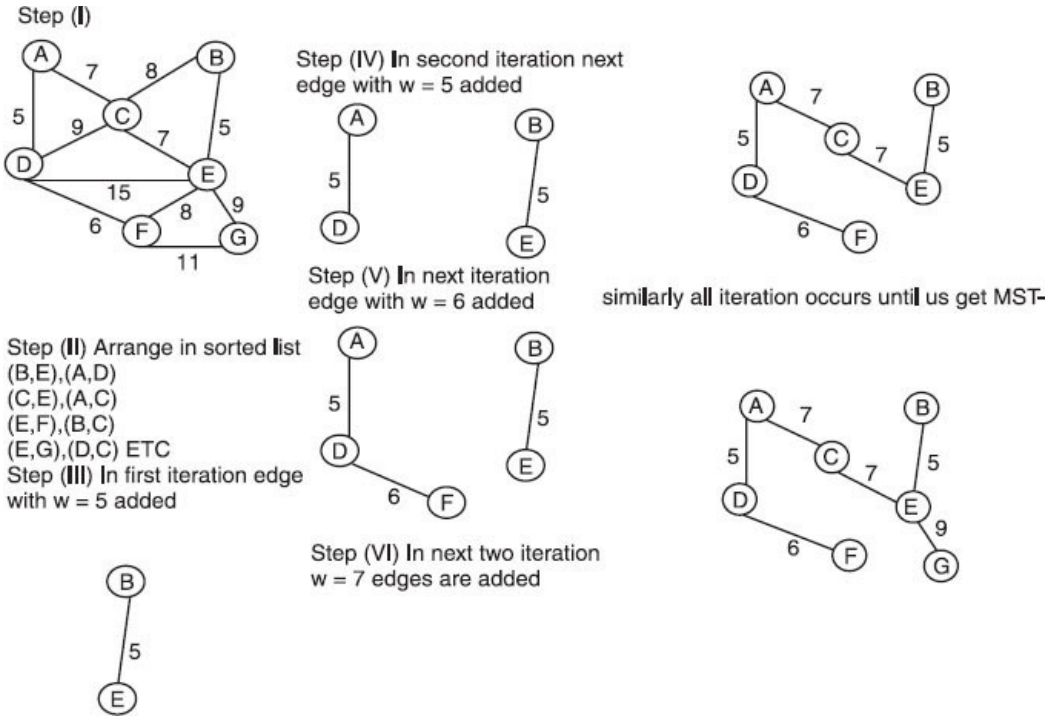
Here in this algorithm-

Make_SET(v): Create a new set whose only member is pointed to by v . Note that for this operation v must already be in a set.

FIND_SET(v): Returns a pointer to the set containing v .

UNION(u, v): Unites the dynamic sets that contain u and v into a new set that is union of these two sets.

Example:



Analysis: The time complexity of MST depends on number of edges and vertexes that it: T(n)=O(E log V)

6.5.3 Prim's Algorithm

This algorithm forms a single tree. The safe edge that is connected to the tree is always a least weighted edge in graph that connects one vertex of the tree to other distinct component without creating a cycle i.e. If an edge (u , v) connect to the tree, then (u) is already added to the set of edges of the MST, and edge (u , v) is the only available least weighted edge.

A little more formally, given a connected, undirected, weighted graph with a function w : E → R.

Starts with each vertex being its own component.

Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them).

Scans the set of edges in monotonically increasing order by weight.

Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

Algorithm

MST_PRIM (G,w,r)

For each u ∈V (G)

Do key[u] ← ∞

π[u] ← NIL

key[r] ← 0

Q ← V [G]

While Q ≠ ∅

dou←Extract_Min (Q)

for each v ∈Adj[u]

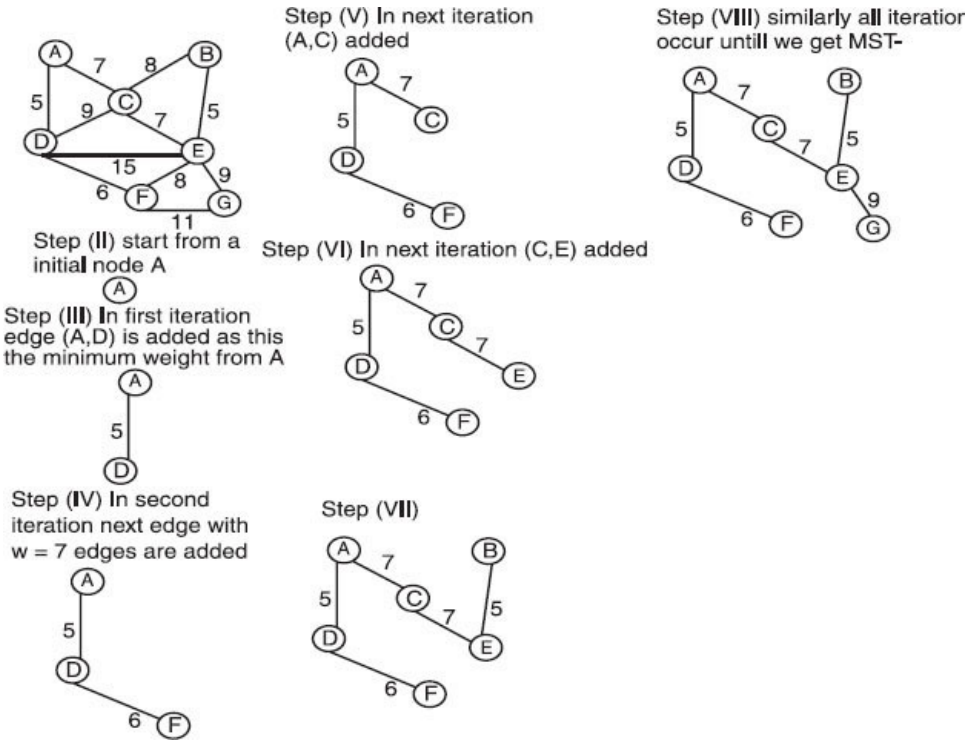
do if v ∈ Q && w(u, v) < key[v]

then π[v] ← u

key [v] ← w(u, v) ► *decrease key*

where Extract_Min return the edge v such that (u,v) has lightest weight in (v, v-V[G]) and key[v] ← ∞ if v is not adjacent to u

Example:



Analysis: The time complexity of MST depends on number of edges and vertexes that's why:

The run time of first for loop is O(V log V)

The while loop executes V times

The Extract_Min executes O(log V)

⇒Extract_Min will execute total O(V log V) times

For loop in line 8-11 will execute O(E)

And decrease key

6.6 Single Source Shortest Path

In single source shortest path algorithm, we need to find the path from starting point such that the traversed path between vertices remain shortest or we may say the total weight of the constituent edges is minimized.

6.6.1 Properties of Single Source

Triangle inequality: For any edge (u, v) ε E, we have δ(s, v) ≤ d(s, u) + w (u, v).

Upper-bound property: d[v]≥δ(s, v) for all vertices v ε V, and once d[v] achieves the value of δ(s, v), it never changes.

No-path property: If there is no path from s to v, then we always have d[v] = δ(s, v) = ∞.

Convergence property: If d [u] be the shortest path in G for some u, v ε V, and if d[u] = δ(s, u) at any time prior to relaxing edge (u, v), then d[v] = δ(s, v) at all times afterward.

Path-relaxation property: If p = is a shortest path from s = v 0 to v k , and the edges of p are relaxed in the order (v 0 , v 1), (v 1 , v 2),…, (v k-1 , v k), then d[v k] = δ(s, v k). This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Predecessor-sub graph property: For d[v] = δ(s, v) for all v ε V, the predecessor sub graph is a shortest-paths tree rooted at s.

6.6.2 Application of Shortest Path Algorithm

Robot navigation.

Texture mapping.

Urban traffic planning.

Optimal pipelining of VLSI chip.

Routing of telecommunications messages

Network routing protocols (OSPF, BGP, RIP).

Optimal truck routing algorithm using given traffic congestion pattern.

6.6.3 Bellman Ford Algorithm

The Bellman ford algorithm solves the single source shortest path problem where edge weights may have negative values. However in general when we use any map to find path, we often not encountered with any negative weights. But in today's world many other problems are supposed to use these algorithms where we should consider negative weights.

The Bellman-Ford Algorithm computes the cost of the cheapest paths from a starting node to all other nodes in the graph. Thus, he can also construct the paths afterwards.

The algorithm proceeds in an interactive manner, with a worst cost estimation in beginning and then improving it iteratively until the correct value is found.

The very first estimate is: The starting node has cost 0, as its distance to itself is obviously be 0.

All other nodes have cost infinity from the starting node, which is the worst estimate possible.

Afterwards, the algorithm checks every edge for the following condition :

Is the cost of the source plus the cost of using the edge smaller than the cost of the target vertex?

If this is the case, the cost of the target gets updated: They are set to the cost of the source plus the cost for using the edge.

Otherwise won't consider it.

Iterate V-1 times to consider all vertices for the same.

The process of looking at all edges of the graph and updating the cost of the nodes is called a phase . Unfortunately, it is not sufficient to look at all edges only once. After the first phase, the cost of all nodes for which the shortest path only uses one edge have been calculated correctly. After two phases all paths that use at most two edges have been computed correctly, and so on.

Algorithm

// v is source node and n is number of nodes // number of edges involved in the path => k=2 means include two edges

// u = 2, 3, 4,5,6,7 // represent all nodes in graph except source

// here i represents the intermediate node

Bellman_Ford(v,cost,dist,n)

for i ← 1 to n

dist[i] = cost[v,i]

for k ← 2 to n

for each u such that u ≠ v and u has at least one incoming edge

do for each in graph

do if dist[u] >dist[i] + cost[I,u]

a. then dist[u] = dist[i] + cost[I,u]

end if

end for

end for

end for

endBellman_Ford

Example:

for i ← 1 to 7

dist[1]=cost[1,1]=0

dist[2]=cost1,2]=6 and so on. As represent in first row of matrix.

for k ← 2 to n

k =2

u = 2, 3, 4,5,6,7 firstly for u =2

(we may have a path using two edges

=> we may take path 1 -> 3 -> 2)

For each <3, 2> in the graph

ifdist[2] >dist[3] + cost[3,2]

6 > 5 -2

dist [2] = 3

for u =3

for each <4,3> in graph

ifdist[3]>dist[4]+cost[4,3]

5 > 5 -2

dist [3]=3

and so on for all remaining u's.

similarly for k=3

for u =2

for each <{3,4},2>

ifdist[2]>dist[4]+dist[3]+cost[3,2] and so on.

Here in the above example k is number of nodes to be used to reach from source to destination.

Here we have taken 1 as source node.

The first row of the matrix is given as a result of edges which are directly connected to source 1 and those which are not directly connected; we represent their distance from source 1 by ∞.

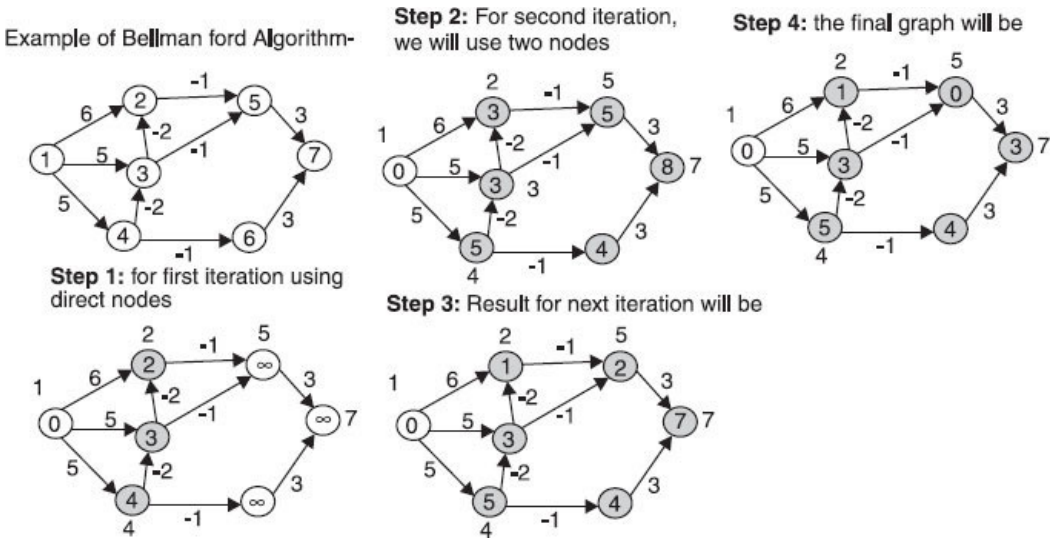
The second row indicate the path from 1 to another node by using some intermediate node i.e. for the path from 1-> 2 we may go using 1-> 3-> 2 and after comparing cost of first path to the second path, we take the lowest cost path.

1 st column represent the distance of node from itself that's why contain all zero i.e. cost of path from 1-> 1.

Similarly 2 nd column represent distance from 1->2 after including one edge that is the distance of indirectly connected nodes, having one intermediate node and so on.

6.6.3.1 Graph with Negative Weights

Perhaps the most important effect is that when negative weights are present low- weight shortest paths tend to have more edges than higher-weight paths. For positive weights, our emphasis was on looking for shortcuts; when the concept of negative weights is present, we seek detours that use as many edges with negative weights as we can find.



Analysis: Time complexity of Bellman-Ford algorithm is Θ(VE) where V is number of vertices and E is number edges. If the graph is complete, the time complexity of E becomes Θ(V ²). So the overall time complexity reaches to Θ(V ³).

6.6.4 Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have positive weights. It uses greedy strategy and is very similar to Prim's algorithm.

Algorithm starts at the source vertex, s, and it grows to form a tree, T, that ultimately spans over all vertices, reachable from S. Vertices are added to T in order of increasing distance from source i.e., first source S is added, then the vertex closest to S, after that the next closest vertex, and so on.

Algorithm

dist[s] ←0 // distance to source vertex is zero

for all v ∈ V-{s}

do dist[v] ∞ // set all other distances to infinity

S ←Φ // S, the set of visited vertices is initially empty

Q ← V // Q, the queue initially contains all vertices

while Q ≠Φ // while the queue is not empty

do u ←mindistance (Q,dist) // select the element of Q with the min. distance

S ← S □{u} // add u to list of visited vertices

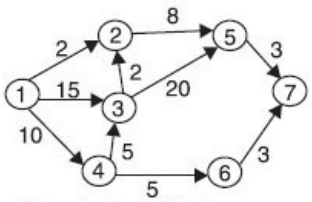
for all v ∈ neighbors[u]

do if dist[v] > dist[u] + w(u, v) //if new shortest path found

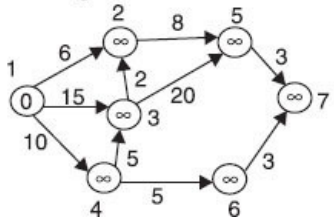
then d[v] ← d[u] + w(u, v) //set new value of shortest path

return dist

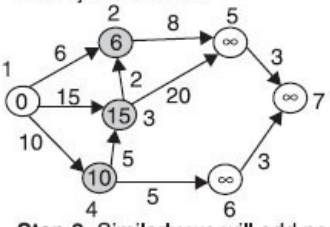
Example of Dijkastra's Algorithm-



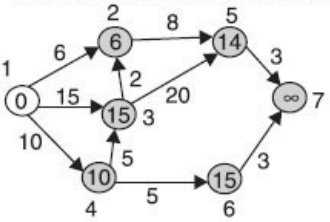
Step 1: Node 1 is the source and we will choose all the nodes, having lowest distancefrom it



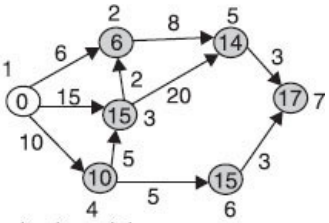
Step 2: Now we will adding all adjecent nodes



Step 3: Similarly we will add nodes having minimum distance in successive iterations.



Step 4: The last step will be



History of Shortest Path Algorithm

Shimbel (1955) Information networks.

Ford (1956) RAND, economics of transportation .

Dantzig (1958) Simplex method for linear programming.

Bellman (1958) Dynamic programming.

Moore (1959) Routing long-distance telephone calls for Bell Labs.

Dijkstra (1959) Simpler and faster version of Ford's algorithm.

6.7 Shortest Paths and Matrix Multiplication (All Pair Shortest Path Problem)

The All-Pairs Shortest Paths problem on a directed graph G = (V , E) uses dynamic approach where each major loop will invoke an operation that is very similar to matrix multiplication.

So we will follow same procedure as we have done in dynamic programming.

Step1: The Structure of a Shortest Path: For the All-Pairs Shortest-Paths problem on a graph G = (V , E), we find a path between vertices such that all sub-paths are shortest paths of given graph. Suppose that the graph is represented by an adjacency matrix W = (w ij). Consider a shortest path p from vertex i to vertex j , and suppose that p contains at most m edges. Assuming that there are no negative-weight cycles, and m is finite.

If i = j , then p has weight 0 and no edges added to the graph.

If vertices i and j are distinct, then we decompose path p into p ij m into p ik m-1 & v k → v j ,

where|p ik m-1 | ≤ m - 1 such that p ik m-1 should be a shortest path from v i to v k by optimal substructure property.

Therefore, δ(v i ,v j) = δ(v i ,v k) + ω kj

Step 2: A Recursive Solution to the All-Pairs Shortest-Paths problem- Let d ij m be the minimum weight of the path from vertex i to vertex j that contains at most m edges.

When m = 0, there is a shortest path from i to j with no edges if and only if i = j .

$$\text{Thus, } d_{ij}^{m} = \begin{cases} 0 & \text{if i} = \text{o} \\ \infty & \text{if i} \neq \text{i} \end{cases}$$

And when m ≥1, we compute as the minimum weight of the shortest path from i to j consisting of at most m - 1 edges and the minimum weight of any path from i to j having at most m edges, obtained by looking at all possible predecessor's k of j .

Thus, we recursively define it as

$$d_{ij}^{m} = \min\{ d_{ij}^{m-1}, \min_{1\le k\le n \wedge k\ne j}\{ d_{ik}^{m-1} + \omega_{kj}\}\}$$
$$= \min_{1\le k\le n}\{d_{ik}^{m-1} + \omega_{kj}\} \text{ for all } v_k \in V,$$

since $\omega_{jj} = 0$ for all $v_j \in V$.

Where the actual weight of the path will be given by δ(v i , v j) = d ij n-1 = d ij n = d ij n+1 since m ≤ n - 1 = | V | - 1

Step 3: Computing the shortest-path weights bottom up: For the given value W = D ¹ , we will compute a series of matrices D ² , D ³ , …, D n-1 ,

where D m = (d ij m) for m = 1, 2,…, n-1

and hence the final matrix Dn-1 contains actual shortest path weights,

i.e., $d_{ij}^{n-1} = \delta(v_i, v_j)$

Algorithm

EXTEND(D , W)

1. D = (d_{ij}) is an n x n matrix
   a. for *i* ← *1* to *n* do
   b. for *j* ← *1* to *n* do
      i. $d_{ij} \leftarrow \infty$
      ii. for *k* ← *1* to *n* do
      iii. $d_{ij} \leftarrow \min\{d_{ij}, d_{ik} + \omega_{kj}\}$
2. return D

Suppose we wish to compute the matrix product $C = A \cdot B$ of two $n \times n$ matrices A and B. Then, for $i, j = 1, 2,...,n$, we compute

MATRIX-MULT(A , B)

►C = (c_{ij}) is an n x n result matrix

for *i* ← *1* to *n* do

for *j* ← *1* to *n* do

$c_{ij} \leftarrow 0$

for *k* ← *1* to *n* do

$c_{ij} \leftarrow c_{ij} + a_{ik} \times b_{kj}$

return C

Note: Here relation to matrix multiplication C = A B : $c_{ij} = \sum_{1\le k\le n} a_{ik} \times b_{kj}$,

- $D^{m-1} \leftrightarrow A$ & $W \leftrightarrow B$ & $D^m \leftrightarrow C$

"min" ↔ "t" & "t" ↔ "x" & "∞" ↔ "0"

Thus, we compute the sequence of matrix products

$D^1 = D^0 \times W = W$; note D^0 = identity matrix,

$D^2 = D^1 \times W = W^2$

$D^3 = D^2 \times W = W^3$

$D^{n-1} = D^{n-2} \times W = W^{n-1}$

$$i.e., Dij^0 = \begin{cases} 0 \text{ if } i = j \\ \infty \text{ if } i \ne j \end{cases}$$

The matrix D n-1 =W n-1 contains the shortest-path weights. The following procedure computes this sequence

SLOW-ALLPAIRSHORTEST_PATH(W)

D ¹ ← W

for m ← 2 to n-1 do

D m ← EXTEND(D m-1 , W)

return D n-1

Analysis: The EXTEND(W) procedure has a complexity = Θ(n ³)

The matrix multiplication also has a complexity = Θ(n ³)

And the SLOW-ALLPAIRSHORTEST_PATH (W) procedure has complexity =Θ(n ⁴), since it is calling extend function in a for loop.

Example:

Step 1: All the distance are taken direct from source to target if there is no directed edge then we take distance as infinite

	1	2	3	4	5
1	0	3	6	8	-4
2	8	0	8	1	7
3	8	3	0	8	8
4	2	8	-5	0	8
5	8	8	8	4	0

D1=D0*W

Step 2: In this step we can add one intermediate node between source and target.

E.g. 1→4 we have two paths

1→5 → 4=4+4==0 &&

1→2 → 4=3+1=4

Since first one is smaller we will consider it.

Similarly for 2 →1 we go from 2→5→1

For 2 →3 we go from 2→4→3

For 3 →4 we go from 3→2→4

For 3 →4 we go from 3→2→4

Same for 4 →2 we go from 4→3→2

For 4→5 we go from 4→1→5

For 5→1 we go from 5→4→1

	1	2	3	4	5
1	0	3	6	0	-4
2	3	0	-4	1	7
3	8	3	0	4	10
4	2	-2	-5	0	-2
5	6	8	-1	4	0

D2=D1*W

Step 3: In this step we can add two intermediate nodes between source and target.

For 1→3 we go from 1→5→4→3

For 2→5 we go from 2→4→1→5

For 3→1 we go from 3→2→4→1

For 5→2 we go from 5→4→3→2

	1	2	3	4	5
1	0	3	-5	0	-4
2	3	0	-4	1	-1
3	6	3	0	4	10
4	2	-2	-5	0	-2
5	6	2	-1	4	0

D3=D2*W

Step 4: In this step we can add three intermediate nodes between source and target.

For 1→2 we go from 1→5→4→3→2

For 3→5 we go from 3→2→4→1→5

	1	2	3	4	5
1	0	-2	-5	0	-4
2	3	0	-4	1	-1
3	6	3	0	4	2
4	2	-2	-5	0	-2
5	6	2	-1	4	0

D4=D3*W

Step 5: If we follows the same steps and add four intermediate nodes from source and target. Then we will get D5=D4

6.8 The Floyd-Warshall Algorithm

In this algorithm we are considering the negative edges, but no negative weights cycle should be considered. Same as in the above algorithm, we will use dynamic approach to develop this algorithm and after finding the final result we will find the transitive closure of a directed graph.

Step 1: The structure of a shortest path: In the Floyd-Warshall algorithm, we use a different characterization of the structure of a shortest path than we used in the matrix-multiplication-based all-pairs algorithms. The algorithm considers the "intermediate" vertices of a shortest path , where an intermediate vertex of a simple path p = is any vertex of p other than v 1 or v k , that is, any vertex in the set {v 2 , v 3 ,…, v k-1 }.

Our assumptions are that the vertices of G are V = {1, 2,…, n}, we consider only a subset {1, 2,…, k} of vertices for some k. For any pair of vertices i, j V, consider all paths from i to j whose intermediate vertices are all taken from the set {1, 2,…, k}, and let p be a minimum-weighted path among them (Path p is simple). The Floyd-Warshall algorithm exploits a relationship between the minimum weighted path p and shortest paths from i to j with all intermediate vertices drawn from the set {1, 2,…, k - 1}. The relationship depends on whether or not k is an intermediate vertex of path p. The Floyd-Warshall algorithm states that:

If k is not an intermediate vertex of path p, then all intermediate vertices of path p will be in the set {1, 2,…, k - 1}. Thus, a shortest path from vertex i to vertex j with all intermediate vertices exists in the set {1, 2,…, k - 1} is also a shortest path from i to j with all intermediate vertices lies in the set {1, 2,…, k}.

If k is an intermediate vertex of path p, then we break p down into two paths, p1 is a shortest path from i to k with all intermediate vertices in the set {1, 2,…, k}. Because vertex k is not an intermediate vertex of path p1, we see that p1 is a shortest path from i to k with all intermediate vertices in the set {1, 2,…, k - 1}. Similarly, p2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set {1, 2,…, k - 1}.

Step 2: A recursive solution to APSP problem: Let d k ij be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1,2,…, k}.

When k = 0, a path from vertex i to vertex j with no intermediate vertex at all. Such a path has at most one edge, and hence d ij º =w ij . A recursive definition following the above discussion is given by

$$d_{ij}^{k} = \begin{cases} w_{ij} & -4 \\ \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}) & -5 \end{cases}$$

Since for any path, all intermediate vertices are in the set {1, 2,…, n}, the matrix D m ≡ (d y m) gives the final result i.e. d ij m = δ(i , j) for all i, jεV.

Step 3: Computing The Shortest Path Weights- The following bottom up approach is used to calculatedistance d m y . For computing matrix W of size n × n , we are using here No Path Property of Single Source Shortest Path.

Algorithm

FLOYD-WARSHALL(W)

n ← rows[W]

D(0) ← W

for k ← 1 to n

do for i ← 1 to n

do for j ← 1 to n

i. do d ij k ← min(d ij k - ¹ , d ik k - ¹ + d kj k - ¹)

return D(n)

Example: We are taking the same example as we have discussed in the above algorithm.

Adjacency matrix will be or we may say D (0) :

Step 1: We will compute for all possible pair

$d_{ij}^{k} \leftarrow \min\left(d_{u}^{k-1},\ d_{k}^{k-1} + k_{kj}^{k-1}\right)$

$d_{11}^{1} = \min\left(d_{11}^{0},\ d_{11}^{0} +\ d_{11}^{0}\right) = 0$

$d_{12}^{1} = \min\left(d^{0}_{12}, d^{0}_{11}, d^{0}_{12}\right)$
$= \min(3, 3) = 3$

Similary $d^{1}_{13} = 6$, $d^{1}_{14} = \infty$ and $d^{1}_{15} = 4$

$d^{1}_{21} = \min\left(d^{0}_{21}, d^{0}_{21} + d^{0}_{11}\right) = \infty$

$d^{1}_{22} = \min\left(d^{0}_{21}, d^{0}_{21}, d^{0}_{12}\right) = \min(0, \infty) = 0$

$d^{1}_{23} = \infty\ d^{1}_{24} = 1$ and $d^{1}_{25} = 7$

$d^{1}_{31} = \min\left(d^{0}_{31}, d^{0}_{30} + d^{0}_{11}\right) = \infty$

$d^{1}_{32} = 3,\ d^{1}_{33} = 0$ and $0\ d^{1}_{34} = d^{1}_{35} = \infty$

$d^{1}_{41} = \min(2, 0 + 2) = 2$

$d^{1}_{42} = \min(\infty\ 2 + 3) = 5$

$d^{1}_{43} = \min(-5, 2 + \infty) = -5$

$d^{1}_{44} = \min(0, 2 + \infty) = 0$

$d^{1}_{45} = \min(\infty, 2 - 4) = -2$

$d^{1}_{51} = \infty$

$d^{1}_{52} = \min(\infty, \infty) = \infty$

$d^{1}_{53} = \infty$

$d^{1}_{54} = \min(4, \infty) = 4$

$d^{1}_{55} = 0$

$\Rightarrow D^{(1)} =$

Step 2: Again we will follow same steps

$d^1_{11} = \min(d^1_{11}, d^1_{11}, d^1_{11}) = 0$

$d^2_{12} = \min(d^1_{12}, d^1_{12}, d^1_{22}) = 0$

$= \min(3, 3 + 0) = 3$

Similary $d^3_{13} = 6$

$d^2_{14} = -4$ $d^2_{13} = 6$,

and $d^2_{15} = -4$ $d^2_{32} = 3 = d^2_{33} = 0$,

$d^2_{24} = 1$ and $d^2_{32} = 7$

Similary $d^2_{31} = \infty$, $d^2_{32} = 3$, $d^2_{33} = 0$

$d^2_{34} = \min(\infty, 3 + 1) = 4$

$d^2_{35} = \min(\infty, 3 + 7) = 10$

$d^2_{41} = 2$, $d^2_{42} = 5$, d^2_{43}, $d^2_{44} = 0$ and $d^2_{45} = -2$

Similary values of fifth row remain same.

$\Rightarrow D^{(2)} =$

Step 3: Now the computed values for D (3) are

$d^3_{11} = 0$, $d^3_{12} = d^3_{13}{}' = 6$, $d^3_{14} = 10$, $d^3_{15} = -4$

$d^3_{21} = \infty$, $d^3_{22} = \infty$ $d^3_{23}{}' = \infty$ $d^3_{24} = 1$, $d^3_{25} = 7$,

$d^3_{31} = \infty$, $d^3_{32} = 3$, $d^3_{33} = 0$, $d^3_{34} = 4$, $d^3_{35} = 10$,

$d^3_{41} = 2$, $d^3_{42} = -2$, $d^3_{43} = 5$, $d^3_{44} = -1$, $d^3_{45} = 2$,

And $d^3_{51} = \infty$, $d^3_{52} = \infty$, $d^3_{53} = \infty$, $d^3_{54} = 4$, $d^3_{55} = 0$

$\Rightarrow D^{(3)} =$

	1	2	3	4	5
1	0	3	6	10	-4
2	∞	0	∞	1	7
3	∞	3	0	4	10
4	2	-2	-5	-1	-2
5	∞	∞	∞	4	0

Step 4: Now the computed values for D (4) are

$d^4_{11} = 0, d^4_{12} = 3, d^4_{13}{}' = 5, d^4_{14} = 9, d^4_{15} = -4$

$d^4_{21} = 3, d^4_{22} = 1, d^4_{23} - 4, d^4_{24} = 0, d^4_{25} = -1,$

$d^4_{31} = 6, d^4_{32} = 2, d^4_{33} = 1, d^4_{34} = 3, d^4_{35} = 2,$

$d^4_{41} = 1, d^4_{42} = -3, d^4_{43} = 6, d^4_{44} = -2, d^4_{45} = 3,$

$d^4_{51} = 6, d^4_{52} = 2, d^4_{53} = -1, d^4_{54} = 3, d^4_{55} = 0$

$\Rightarrow D^{(4)} =$

	1	2	3	4	5
1	0	3	5	9	-4
2	3	-1	-4	0	1
3	6	3	-1	3	2
4	1	-3	-6	-2	-3
5	6	2	-1	3	0

Step 5: Now the computed values for D (5) are

$d^5_{11} = 0, d^5_{12} = -2\ d^5_{13} = -5, d^5_{14} = 1, d^5_{15} = -4$

$d^5_{21} = 3, d^5_{22} = -1\ d^5_{23} = -4, d^5_{24} = 0, d^5_{25} = -4$

$d^5_{31} = 6, d^5_{32} = 2\ d^5_{33} = -1, d^5_{34} = 3, d^5_{35} = -2,$

$d^5_{41} = 1, d^5_{42} = -3\ d^5_{43} = -6, d^5_{44} = 3, d^5_{45} = -3$

$d^5_{41} = 1, d^5_{42} = -3\ d^5_{43} = -6, d^5_{54} = 3, d^5_{55} = 0$

$\Rightarrow D^{(5)} =$

	1	2	3	4	5
1	0	3	5	9	-4
2	3	-1	-4	0	1
3	6	3	-1	3	2
4	1	-3	-6	-2	-3
5	6	2	-1	3	0

REVIEW EXERCISE

What do you mean by graph? Write and explain any two real world applications of graph.

What is Dijkstra's algorithm for finding the shortest path in a graph with suitable example?

Compare the performance of various graph theoretic algorithms in terms of time complexity?

How can we compute the shortest path using Bellman-ford algorithm with negative weights?

What is the difference between Prim's and Krushkal's algorithm?

MULTIPLE CHOICE QUESTIONS & ANSWERS

Consider the tree T arcs of a BFS traversal from a source node W, which is unweight, connected and undirected graph as well. The tree T used to compute:

(a) the shortest path between every pair of vertices.

(b) the shortest path from W to every vertex in the graph.

(c) the shortest paths from W to only those nodes that are leaves of T.

(d) the longest path in the graph

Ans. (b) the shortest path from W to every vertex in the graph.

BFS always produces shortest paths from source to all other vertices in an unweighted graph. In BFS, we first explore all vertices which are immediate from source, then we explore all vertices which are having one intermediate edge away from the source and so on. This property of BFS makes it useful in many algorithms like Edmonds-Karp algorithm

Let G be an undirected graph. Consider a depth-first search traversal of graph G, and T be the resulting depth-first search tree. Let u be a vertex in G and let v be the unvisited vertex, visited after visiting u in the traversal. Which of the following statements is always true?

(a) if {u,v} represnts an edge in G, then u is a descendant of v in T

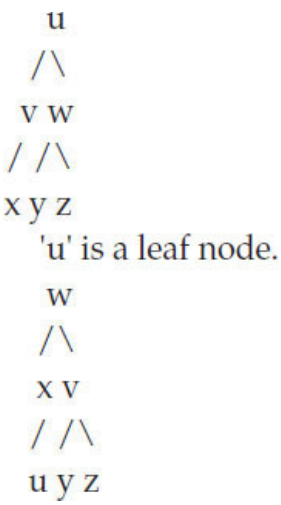
(b) if {u,v} represent an edge in G, then v must be a descendant of u in T

(c) If {u,v} is not an edge in G then u is a leaf in T (GATE CS 2000)

(d) If {u,v} is not an edge in G then u and v must have the same parent in T

Ans. (c) If {u,v} is not an edge in G then u is a leaf in T

Explanation: In DFS, if 'v' is visited after 'u', then one of the following is true. (u, v) is an edge.



In DFS , after visiting a node, we first traverse for all unvisited children. If there is no unvisited child (u is leaf), then control goes back to parent and parent then visits next unvisited children.

Which of the following statement(s) is / are correct for Bellman-Ford shortest path algorithm?

P: Always finds a negative weighted cycle, if one exists.

Q: Finds whether any negative weighted cycle is reachable from the source.

(a) P Only

(b) Q Only

(c) Both P and Q

(d) Neither P nor Q

Ans. (b) Q Only

Explanation: Bellman-Ford shortest path algorithm is a single source shortest path algorithm. So it can only find cycles which are reachable from a given source, not any negative weight cycle

If there exists a negative weight cycle, then it will surely appear in shortest path. As the negative weight cycle will always form a shortest path when iterated through the cycle again and again.

Consider the following graph

Which are depth first traversals of the above graph among the following sequences-

I) a b e g h f

II) a b f e h g

III) a b f h g e

IV) a f g h b e (GATE CS 2003)

(a) I, II and IV only

(b) I and IV only

(c) II, III and IV only

(d) I, III and IV only

Ans. (d) I, III and IV only

Explanation: We can check all DFSs for following properties as discussed in question 2.

Let G = (V, E) be an undirected graph with a sub-graph G1 = (V1, El). Weights are assigned to edges of G according to the below expression:

$$w(e) = \begin{cases} 0 \; if \; e \in \mathrm{E1} \\ 1 \; otherwise \end{cases}$$

A single-source shortest path algorithm is executed on the weighted graph (V, E, w) with an arbitrary vertex v1 of V1 as the source. Which of the following can always be inferred from the path costs computed?

(a) The number of edges in the shortest paths from v1 to all vertices of G

(b) G1 is connected (GATE CS 2003)

(c) V1 forms a clique in G

(d) G1 is a tree

Ans. (b) G1 is connected

Explanation: When shortest path from v1 (one of the vertices in V1) is computed. G1 is connected if the distance from v1 to any other vertex in V1 is greater than 0, otherwise G1 is disconnected.

Consider the DAG with Consider V = {1, 2, 3, 4, 5, 6}, shown below. Which of the following is NOT a topological ordering?

(a) 1 2 3 4 5 6

(b) 1 3 2 4 5 6

(c) 1 3 2 4 6 5

(d) 3 2 4 1 6 5 (GATE CS 2007)

Ans. (d) 3 2 4 1 6 5

Explanation: 1 appears after 2 and 3 which is not possible in Topological Sorting.

Let G = (V, E) be a simple undirected graph, and s be a particular vertex in it called the source. For x ← V, let d(x) denotes the shortest distance in G from s to x. A breadth first search (BFS) is performed starting at node s. Let T be the resultant BFS tree. If (u, v) is an edge of G that is not in T, then which one of the following cannot be the value of d(u) - d(v)?

(a) -1

(b) 0

(c) 1

(d) 2 (GATE CS 2015)

Ans. (d) 2

Explanation: Note that the given graph is undirected, so an edge (u, v) also means (v, u) is also an edge.

Since a shorter path can always be obtained by using edge (u, v) or (v, u), the difference between d(u) and d(v) can not be more than 1.

Suppose depth first search is executed on the graph below starting at some unknown vertex. Assume that a recursive call to visit a vertex is made only after first checking that the vertex has not been visited earlier. Then the maximum possible recursion depth (including the initial call) is____.

(a) 17

(b) 18

(c) 19

(d) 20 (GATE CS 2014)

Ans. (c) 19

Explanation: The following diagram shows the worst case situation where the recursion tree has maximum depth.

So, the recursion depth is 19 (including the first node).

C HAPTER -7

Backtracking Algorithms

In this chapter student will understand:

What is the backtracking method for problem-solving? How is it different from other strategies? what are the main application areas for backtracking?

7.1 Backtracking Algorithm

Backtracking is a form of recursive operations. As the name suggests we backtrack over previous path to find the solution. Here, We start with one possible move out of many available possibilities and try to solve the problem if we are able to solve the problem with the selected move then we will get the final result. Otherwise, we will backtrack to a previous state and select some other option and again try to solve it. If none of the moves work out, we will claim that there is no end solution for the problem.

The name backtrack was first coined by D. H. Lehmer in the 1950's . Early workers who studied the process were R. J. Walker who gave an algorithmic account of it in 1960 and Golomb and Baumert who presented a very general description of backtracking coupled with a variety of applications.

7.1.1 Generalized Algorithm of Backtracking

Pick a starting point.

While (Problem is not solved)

For each path from the starting point.

check if selected path is safe, if yes select it

Make recursive call to the rest part of the problem

If recursive calls return true, then return true.

Else backtrack the current move and return false.

End For

If no move works out, then return false, NO SOLUTON.

To apply backtracking, The desired solution must be expressible as n tuple (x ¹ , x 2 ,----x n) where x i are chosen from some finite set of solutions S i . The problem is to find a vector which satisfies the criterion function P.

Suppose m I is size of set S i , then m= m 1 m 2 ……m n are n tuples which are possible candidates to satisfy the function P.

The basic idea behind this is to build up one vector one component at a time and use modified criterion function P i (x 1 ,x 2 ,----x n) called bounding function to test whether vector has any chance of success or not. The major advantage of this method is this: if it is realized in early stages that the partial vector (x1, x2,……., x;) cannot lead to an optimal solution, then rest possible test vectors (m+ 1 • • • mn) may be ignored entirely.

7.1.2 Constraints

Backtracking requires that a solution must satisfy a complex set of constraints where constraints can be of two types:

Explicit constraints: Explicit constraints are rules which restrict each x; to take on values only from a given set. Such as:

$x_i \geq 0 \ \forall \ S_i = \{nonnegativevalues\}$
$x_i = \text{or } 1 \ \forall \ S_i = \{0,1\}$

The explicit constraints might rely upon the particular instance I of the problem being settled. All tuples that fulfill the explicit constraints characterize a conceivable solution for that instance I.

Implicit constraints: The implicit constraints figure out which of the tuples in the solution space of instance I fulfill the criterion functions. In this manner implicit constraints depict the path in which the x; must relate to each other.

Example: We use backtracking in playing many games like Sudoku, solving word puzzles, playing chess, finding path in a maze, tug of war etc. Where we use all possibilities to solve the problem. Backtracking has a similar concept. Here we will examine it with Sudoku.

Given a partially filled 9×9 2D array 'grid[9][9]' the objective is to dole out digits (from 1 to 9) to the vacant cells so that every row, column, and sub grid of size 3×3 contains exactly one instance of the digits from 1 to 9.

Here explicit conditions for sudoku are:

Partially filled 9×9 2D array

Digits (from 1 to 9)

Subgrid of size 3×3

And implicit constraints for Sudoku are

Assign digits (from 1 to 9) to the empty cells of every row, column

Subgrid of size 3×3 contains exactly precisely one case of the digits from 1 to 9.

We be that a similar number is absent in current line, current segment and current 3X3 sub grid. In the wake of checking for safety, we allot the number, and recursively check whether this assignment leads to a solution or not. On the off chance that the task doesn't prompt an answer, at that point we attempt next number for current exhaust cell. And if the task doesn't prompt an answer, i.e. if none of number (1 to 9) lead to solution, we return false.

Step 1: Firstly we should try to fill the grids according to the constraints imposed. Here we have completely filled one column and one sub grid containing all digits from 1-9.

Step 2: Similarly we fill one row containing values from 1-9.

Here the data highlighted in red is representing erroneous data.

Step 3: If we fill mistakenly some data i.e. duplicity of data in rows, column or in diagonal. Then we have to backtrack it till the safest position and then we further try to solve this.

	9	1	4	2			5	
	2	5		9				4
4	6	7	1	3	5	2	8	9
5	3	4	8		7			
1	8	2		4		5	7	6
6	7	9	5	1	2	3	4	8
9	5	3		8		4		
7	4	6		5		8	9	
2	1	8			4			

Step 4: When all grids fill without any error. Then we get the resultant grid.

8	9	1	4	2	6	7	5	3
3	2	5	7	9	8	1	6	4
4	6	7	1	3	5	2	8	9
5	3	4	8	6	7	9	1	2
1	8	2	3	4	9	5	7	6
6	7	9	5	1	2	3	4	8
9	5	3	6	8	1	4	2	7
7	4	6	2	5	3	8	9	1
2	1	8	9	7	4	6	3	5

Algorithm for backtracking:

Procedure BACKTRACK(n)

// All solutions are generated in X(l:n) and printed as soon as they are

// determined. T(X(l), …, X(k - 1)) gives all possible values of//

// X(k) given that X(l), …, X(k - 1) have already been chosen

// The predicates B k(X(l), …, X(k)) determine those//

// elements X(k) which satisfy the implicit constraints.

integer k, n; local X(l:n)

k ← 1

while k >0 do

if there remains an untried X(k) such that

X(k)ε T(X(l), …,X(k - l)) and B k (X(l), …,X(k)) = true

then if (X(l), …, X(k)) is a path to an answer node

i. then print (X(l), …, X(k))

ii. endif

k ← k + 1 // consider the next set/ /

else k←k - 1 //backtrack to previous set//

endif

repeat

end BACKTRACK

T(X(k)) will yield the set of all possible values which can be placed as the first component, X(l), of the solution vector. X (l) will take on those values for which the bounding function B 1(X(l)) is true. Also note how the elements are generated in a depth first manner. k is continually incremented and a solution vector is grown until either a solution is found or no untried value of X(k) remains. When k is decremented, the algorithm must

resume the generation of possible elements for the kthposition which have not yet been tried.

7.2 N Queen Problem

Considers an n x n chessboard and try to find all ways to place n non attacking queens. We can let (X1, …,Xn) represent a solution, where X is the column of the i th row where the ith queen is placed. The X i S will all be distinct since no two queens can be placed in the same column.

If we imagine the squares of the chessboard being numbered as the indices of the two dimensional array A(l:n, l:n) then we observe that for every element on the same diagonal which runs from the upper left to the lower right, each element has the same "row - column" value. Also, every element on the same diagonal which goes from the upper right to the lower left has the same "row + column" value. Suppose two queens are placed at positions (i,j) and (k, /). Then by the above they are on the same diagonal only if

i - j = k - I or i + j = k + I.

The first equation states that

j-l=i - k

On the other hand the second equation states

j-l=k - i

Therefore two queens lie on the same diagonal if and only if | j-l |= | k-i |

Algorithm:

Procedure PLACE (k) returns a Boolean value which is true if the k th queen can be placed at the current value of X(k). It tests both if X(k) is distinct from all previous values X(l), …, X(k - 1) and also if there is no other queen on the same diagonal.

Procedure PLACE(k)

```
//  X is a global array whose first k values have been set.//
//  ABS(r) returns the absolute value of r//
 1. global X(l: k ); integer i, k
    a. for i ← 1 to k
            i.   do if X(i)= = X(k)                    //two in the same column//
            ii.  or ABS(X(i) – X(k)) = ABS(i – k) //in the same diagonal//
            iii. then retum(false)
         1.           endif
           iv.        repeat
 2. return(true)
 3. end PLACE
```

We can refine the backtracking method with the below given procedure

II. Procedure PlaceNQUEENS(n)

```
//using backtracking this procedure prints all possible placements of n Queens//
1. integer k, n, X(l:n)
2. X(l) 0; k 1                    //k is the current row; X(k) the current column//
3. while k > 0                    //for all rows do//
4. do X(k)←X(k) + 1 //move to the next column//
   a. while X(k)≤ n and not PLACE(k) //can this queen be placed?//
   b. do X(k)←X(k) + 1
   c. repeat
5. if X(k) ≤ n                    //a position is found//
   a. then if k == n              //is the solution complete? //
   b.
        i.   then print(X) // print the array//
        ii.  else k←k + 1;
        iii. X(k) ←0 //go to the next row//
        iv.  endif
   c. else k ←k – 1               //backtrack//
   d. endif
6. repeat
7. endPlaceNQUEENS
```

For Place (k) computing time complexity is O(k - 1).

Analysis: for function Place (k) time complexity is O(n)

For the function PlaceNQueens() time complexity -

In PlaceNQueens() loop is iterating n times and each iteration it invokes the function Place (k) => time complexity O(n2)

Again after placing a queen, we will iterate for all possible positions of leftover queens which in turn result in n × T(n-1) times

⇒ After combining this the total complexity of N queens Problem is T(n)= n × T(n-1)+ O(n2)

Which after solving result in n3+ n! => final complexity = O(n!)

7.2.1 The Four Queen Problem

We can find solution for 2 queens, 4 queen, 8 queen, 16 queen etc., problems with the help of generalized N queen algorithm. Let's have a chance to begin talking about it with a case of four queens where we have an array of size 4 ×4. In chess, a queen can move as far as she pleases, horizontally, vertically, or diagonally. Here chess board has 4 rows and 4 columns. The question is the most effective method to put 4 queens on an ordinary chess board so none of them can hit some other in one move.

Here explicit conditions are:

We have four queens Q1, Q2, Q3 and Q4

We have a 4×4 order matrix

Implicit conditions are

No two queens can be placed in same row as well as in same column.

No two queens can be placed in same diagonal position.

Step 1: Place first queen in the chessboard. Start placing queens from the lowest left corner.

Q1			

Step 2: Now, we will place second queen. Keep in the mind that it can't be placed in the same row, same column as well as in the same diagonal.

×	×	Q2	
Q1			

Step 3: Similarly, place third queen and the forth one.

×	×	×	×
×	Q3	×	×
×	×	Q2	
Q1			

Step 4: As, we can't place forth queen, so we need to do backtracking.

For this we have to place Q3 again. Since we don't find any place for Q3 other than this, we have to backtrack it to the next stage i.e. (Q2).

×	×		Q2
Q1			

Step 5: Again try to place Q3 and Q4 on the chessboard.

×	×	×	×
×	Q3	×	×
×	×		Q2
Q1			

Step 6: With this arrangement, we got no solution. Since we have backtracked for all possible positions of Q2 then we have to replace Q1.

	Q1		

Step 7: Now we will try to place remaining queens.

×	×	Q4	×
Q3	×	×	×
×	×	×	Q2
	Q1		

Hence, this is the resultant positions.

7.3 Sum of subsets

Suppose we are given n distinct positive numbers (usually called weights) furthermore, we want to discover all combinations of these numbers whose sum is M. This is called the sum of subsets problem. In this case the element X(i) of the solution vector is either one or zero depending upon whether the weight W(i) is included or not.

The children of any node are generated by the method that at level i the left child corresponds to X(i) = 1 and the right to X(i) = 0.

A simple choice for the bounding functions is B k (X(l), …, X(k)) = true iff

$$\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i) \geq M$$

Clearly X(l), …, X(k) cannot lead to an answer node if this condition is not satisfied.

The bounding functions may be strengthened if we assume the W(i)s are initially in non-decreasing order. In this case X(l), …,X(k) cannot lead to an answer node if

$$\sum_{i=1}^{k} W(i)X(i) + W(k+1) > M$$

⇨ For X(k)=1 we must have $\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i) > M$

Algorithm

PROCEDURE SUMOFSUB(s, k, r)

//find all subsets of W(l:n) such that sum of elements lead to M. //

//and $r = \sum_{i=1}^{n} W(i)$

The W(j)s are arranged in non-decreasing order.//

//It is assumed that W(1) ≤ M and $\sum_{i=1}^{n} W(i) \geq M$ *//*

global integer M, n;

global real W(1:n); global Boolean X(1:n)

real r, s; integer k, j

// generate left child. s + W(k) ≤ M because B k- 1 = true//

1. X(k) ←1
2. If s + W(k) = =M //subset found//
   a. then print (X(j), j ←1 to k) //there is no recursive call here as W(j) > 0 ∀ 1≤j ≤n //
3. else if s+ W(k) + W(k + 1) ≤ M //B_k = true//
   i. then call SUMOFSUB (s + W(k), k + 1, r – W(k))
      a. end if
4. end if

   //generate right child and evaluate B_k//
5. If s + r – W(k) ≥ M && s + W(k + 1) ≤ M // B_k= true/ /
6. then X(k) 0
   a. call SUMOFSUB(s, k + 1, r – W(k))
7. end if
8. end SUMOFSUB

Procedure SUMOFSUB avoids computing $\sum_{i=1}^{k} W(i)X(i)$

and $\sum_{i=k+1}^{n} W(i)$

each time by keeping these values in variables s and r respectively. The algorithm assumes W(l) ≤ M and $\sum_{i=1}^{n} W(i) \geq M$

. The initial call is call SUMOFSUB $(0, 1, \sum_{i=1}^{n} W(i))$

then X(k + 1), …, X(n) must be zero. These zeros are omitted from the output. Here we have not test for $\sum_{i=1}^{k} W(i)X(i) + \sum_{i=1}^{n} W(i) \geq M$

as we already know s + r≥M and X(k) =1.

Analysis: the time complexity of sum of the subset is- O(2n×N)

Example: For the instance, we have n = 5, number of weights,such that (w1, w2, w3, w4, w5) = (11, 6, 12, 24, 8) and need to find a subset of weights such that total weight M = 30. Find all possiblesum of subsets.

Here the rectangular nodes list the values of s, k, r on each of the calls to SUMOFSUB. Circular nodes represent points at which a subset with sum M is printed out.

Step 1: call SUMOFSUB(0, 1,61)

X(1) ←1

If s + W(1)≠ 30

else if s+ W(1) + W(2) ≤ 30

then call SUMOFSUB(0+11, 2, 50)

Since this is a call for left node that's why the resultant will be

Step 2: call SUMOFSUB(11, 2, 50)

X(2) ←1

If 11+6 ≠ 30

else if 11+6+12 ≤ 30

then call SUMOFSUB(11+6, 3,44)

Step 3: Similarly, we will call for W3. We got

Step 4: Here we find that we cannot traverse in the same node as adding more weight will result in a w> 30.

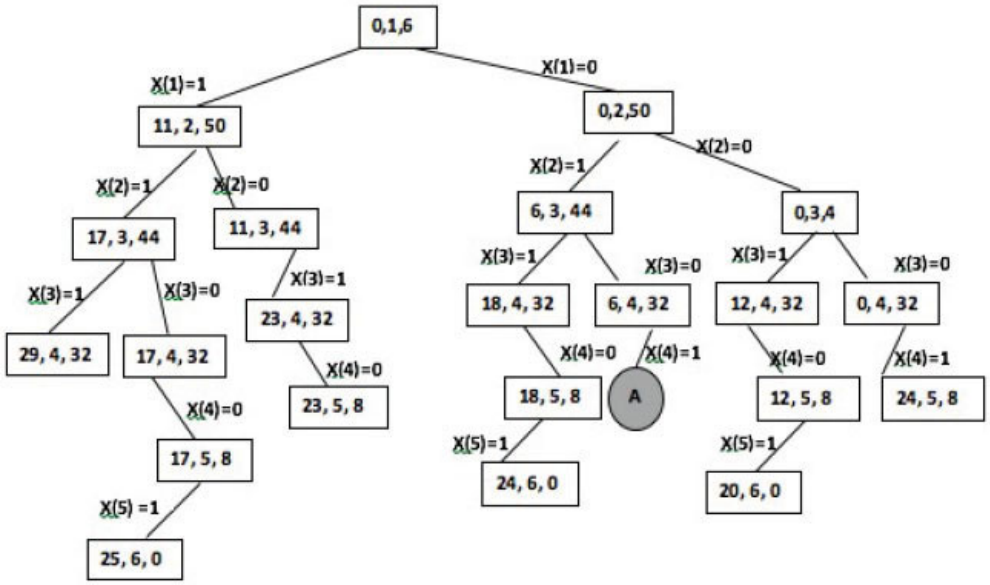
That's why we backtrack to the previous node and add a right leaf to it, which means we are by passing w3 and now considering w4.

Step 5: We came to stage where we can't add w4 but can consider w5.

Step 6: Again we have to backtrack to a safer state that will be x(2)=0.

This process will remain continue in the same manner till we find all possible combinations of weights to have the total weight equal to 30.

Step 7: The resultant tree created by the algorithm SUMOFSUBSET is given beneath. Here we get just a single acceptable state signified by A, with state (0, 1, 0, 1). Here we have iterated only 24 rectangular nodes or may say 24 states while the complete SUMOFSUBSET tree must have total 2 n -1=2 ⁵ -1=32-1=31 rectangular nodes.



7.4 Graph Coloring

Graph coloring is a special case of graph labeling problem; where we can make assignment of labels, may be refer as "colors" to the elements of a graph, which are subjected to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph in such a way that no two adjacent vertices share the same color; this method is called a vertex coloring . Similarly, an edge coloring allots shading to each edge with the end goal that no two contiguous edges share a similar shading, and a face coloring of a planar graph appoints a color to each face or region so that no two faces that share a common boundary will have the same color. Vertex coloring is the starting point of any coloring problem, and other coloring problems can be transformed into a vertex coloring problem.

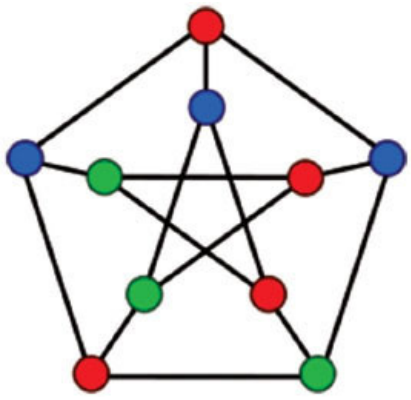
7.4.1 m Coloring Problem

Given an undirected graph G and a number m, which characterizes that the graph can be colored with at most m colors such that no two nearby vertices of the chart are hued with same shading. Here coloring of a graph implies a task of colors to all vertices of that graph.

Input:

A 2D array graph G[V][V] where V is the number of vertices in graph and G[V][V] is adjacency matrix representation of the graph. A value G[i][j] is 1 if there is a direct edge from i to j, otherwise the assignment of value to G[i][j] is 0.

An integer m which is maximum number of colors that can be used. Output: An array color[V] that should have numbers from 1 to m. color[i] should represent the color assigned to the i th vertex. The code should also return false if the graph cannot be colored with m colors.



Algorithm

if all colors are assigned

print vertex assigned colors

else

Try all colors and doled out a shading to the vertex

If color task is conceivable, assign a color to next vertices

If shading task is impractical, de-allot hues and return false

Procedure m_coloring (i:integer)

var color: integer;

begin if (promising (i)) then

if (i = n)

then Write vcolor [1] through vcolor [n];

(a) else

for color = 1 to m // Try every

begin vcolor [i + 1] = color;

m_coloring (i + 1); // next vertex.

end;

end;

Analysis

The time complexity of m coloring is O(nm n)

Example

Consider the graph in figure below. There is no solution to the 2-Coloring problem for this graph because, if we can allot at maximum two different colors, there is no way to color the vertices so that no adjacent vertices are the same color.

In this graph we have 4 nodes There are six solutions to the 3-Coloring problem as shown in the table below:

Nodes	SOLUTIONS					
	Color	Color	Color	Color	Color	Color
V1	1	2	2	3	3	1
V2	2	1	3	2	1	3
V3	3	3	1	1	2	2
V4	2	1	3	2	1	3

7.5 Hamiltonian Cycle

The problem to find a Hamiltonian cycle or path in a given undirected graph is a special case of The Travelling Salesman Problem. The Hamiltonian circuit is named after Sir William Rowan Hamilton . Hamiltonian Path in an undirected graph is a path that traverses each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there exists an edge in graph from the last vertex to the first vertex.

The constraints for generating cycle:

Each vertex must be gone to at any rate once

Any edge can be used only once

Some of the edges can be skipped, but vertices can't be skipped.

If it contains a way between each pair of vertices with an edge between them, it is considered to have distance 1, while if there is no edge between vertex pairs they are assumed to be separated by distance ∞.

Algorithm (Finding all Hamiltonian cycle)

Algorithm Hamiltonian (k)

Loop

Next value (k)

If (x (k)= =0) then return;

If k= =n then

Print (x)

Else

Hamiltonian (k+1);

End if

Repeat

Algorithm Nextvalue (k)

Repeat

X [k]=(X [k]+1) mod (n+1); //next vertex

If (X [k]=0) then return;

If (G [X [k-1], X [k]] ≠ 0) then

For j=1 to k-1 do

if (X [j]==X [k]) then break// Check for distinction.

If (j==k) then //if true then the vertex is distinct.

If ((k

Until (false);

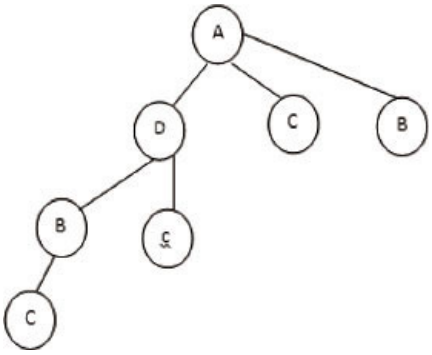
Analysis: In worst scenario of Hamiltonian cycle, in each recursive call one of the remaining vertices is selected. This may result in the decrement of branch factor by 1. Recursion in this case can be thought of as execution of n nested loops where in each loop the number of iterations decreases by one. Hence the time complexity of Hamiltonian Cycle is given by:

T(N) = N(T(N-1) + O(1))*

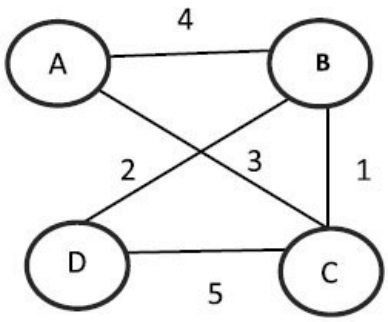
T(N) = N(N-1)(N-2).. = O(N!)

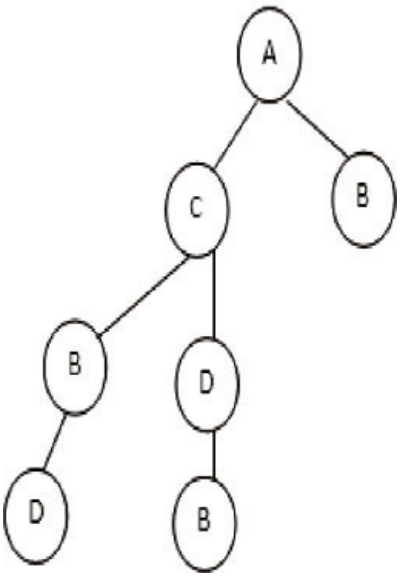
Example: Find the Hamiltonian cycle for the given undirected graph-

If we have vertex A as a source node then we can traversed the graph in the following manner:



As shown in above graph, there are total 3 edges connected from vertex A. Here we considered the minimum weighted edge push out of all conceivable outcomes however when all is said in done we can consider in an indistinguishable request from they show up in the adjacency matrix. Presently we begin from vertex D and afterward go for alternate edges. Here we can't consider the edge lies between D to A on the ground that A is the parent of the node D and can consider both nodes B and C. In the event that we begin from vertex B then node A and D are the ancestors for node B so just a single edge C stay to contemplate. What's more, at node C we had total three connected edges and an edge between nodes C to node Alikewise exists. That is the manner by which, we found the Hamiltonian cycle.





On the off chance that we consider the above chart with a change that there is no edge amongst D and A then the diagram will be, as shown in the

above example of graph in this we can see the use of backtracking algorithm and how we can apply it for determining the Hamiltonian cycle. As shown in figure above, We will start from vertex as assuming that it is the starting vertex. Now there is two edges out of one is to node C and the other is to node B. So, we can start from any vertex. Let's start from vertex C. In algorithm later, we consider the vertex first which come in the matrix notation first. As a recursive call we reach at node D, but we don't have E (D, A) exist so we can have Hamiltonian path but not the Hamiltonian cycle. Again we had to backtrack to B, now B don't have any edge remaining, so again backtrack to C and continued with child node D. Then A-C-D-B-A is the HC. There may be many HC possible in a given graph.

REVIEW EXERCISE

What is Backtracking?

Explain 8-Queen's problem with the help of backtracking algorithm?

Explain the graph coloring algorithm?

How sum of subset problem can be solved using backtracking? Also compare its efficiency by solving it using dynamic programming.

What is Hamiltonian cycle meant for? How it is applied on a graph? Show with the help of an example.

MULTIPLE CHOICE QUESTIONS & ANSWERS

The Hamiltonian cycle's problem uses the following line of code to generate a next vertex, provided x[] is a global array and kth vertex is under consideration:

(a) x[k] ← (x[k] + 1) mod n

(b) x[k] ← (x[k]) mod (n)

(c) x[k] ← (x[k] + 1) mod (n+1)

(d) x[k] ← x[k+1] mod n

Ans: (c) x[k] ← (x[k] + 1) mod (n+1)

The graph colouring algorithm's time can be bounded by _

(a) O(mn m)

(b) O(n m)

(c) O(nm.2n)

(d) O(nmn)

Ans: (c) O(nm. 2n).

Match the following:

(P) Prim's algorithm to find minimum spanning tree	(i) Backtracking
(Q) Floyd-Warshall algorithm for all pairs shortest paths	(ii) Greedy method
(R) Mergesort	(iii) Dynamic programming
(S) Hamiltonian circuit	(iv) Divide and conquer

(a) P-iii,Q-ii, R-iv, S-i

(b) P-i, Q-ii, R-iv, S-iii

(c) P-ii, Q-iii, R-iv, S-i

(d) P-ii, Q-i, R-iii, S-iv

Ans (c) P-ii, Q-iii, R-iv, S-i

Which of the following is not a backtracking algorithm?

(a) M coloring problm

(b) N queen problem

(c) Tower of Hanoi

(d) Knight tour problem

Ans (c) Tower of Hanoi

C HAPTER -8

Branch and Bound

In this chapter student will understand:

What is branch and bound method? How it is implemented? Why it is better than other techniques? The application area of branch and bound in real world scenario.

8.1 Branch and Bound

The general technique for branch and bound algorithms include displaying the solution space as a tree and after that traversing the tree exploring the most promising sub-trees to start. This is preceded until either there are no sub-trees into which we can further break the problem, or we have landed at a point where, if we proceed, just substandard arrangements will be discovered. A generic algorithm for branch and bound searching is introduced below:

Algorithm

Search (t, r, best)

where Pre: t = solution space tree

r = vertex in t

best = best solution found so far

post: best = best solution found after searching sub-tree rooted at r.

If r is a complete solution more optimum than best then set best = r

Generate the children of r.

Compute bounds for vertices in sub-trees of children v 1 , v 2 ,-----,v n = feasible children with good lower bounds

For i = 1 to n

if vi has a promising upper bound

then search (t, vi, best)

The above algorithm can be understood better by following below steps which are actually faced while we implement a complete problem in B & B scenario:

Step 1: Problem instances

In case of graphical problems, like travelling sales person, there is a graph which is represented as an adjacency matrix, or as an adjacency edge list. However, in case of knapsack there is a list of weight of items, another list for their values and an integer value for capacity.

Step 2: Solution tree

This solution tree contains all the feasible, semi-feasible and infeasible solutions of the problem. In case of knapsack problem we create a depth first search tree using objects ordered by weight.

Step 3: Solution Candidates

The solution candidates contain the main elements in the final feasible solution to the problem. For example, in knapsack the final elements of knapsack are the solution candidates.

A fundamental rule to be followed in characterizing solution spaces for branch and bound algorithms is the following:

" If a solution tree vertex is not part of a feasible solution, at that pointthe sub-tree for which it is the root can't contain any feasible solutions."

Lower bound at a vertex: The base estimation of the target function for any node of the sub-tree established at the vertex.

Upper bound at a vertex: The maximum estimation of the objective function for any node of the sub-tree rooted at the vertex.

8.2 The Knapsack Problem

As we have discussed earlier the 0/1 knapsack problem in greedy and dynamic technologies. Now we are representing the same problem in B & B method.

Let us explore all methodologies for this problem.

A Greedy approach is to choose the items from a given set in decreasing order of value per unit weight. The Greedy approach works just for fractional knapsack problem and may not deliver optimal result for 0/1 knapsack.

We can use D ynamic P rogramming (DP) for 0/1 Knapsack problem. In DP, we use a 2D table of size n × W. But in general DP Solution doesn't work well if item weights are not integers .

We can use Backtracking by using the tree representation; we can do DFS of tree. If we reach a point where a solution no longer is feasible, there is no need to continue exploring.

We can even do much better than backtracking if we know the best possible solution sub-tree rooted with every node. If the best in sub-tree is worse than current best, we can simply ignore this node and its sub-trees. So we compute upper bound (best solution) for every node and compare the bound with current best solution before exploring the node.

Example: Consider of size M and select from a set of n objects, where the i th object has weight wi and value pi, a subset of these items to optimize the value contained in the knapsack with the substances of the knapsack not exactly or equivalent to M.

Note: Here we illustrate the method by using tree illustration by using backtracking method, where each node has the value cost and weight for that node and the upper bound of cost and edge illustrate whether we are adding that weight or not by using 1 and 0 notation resp.

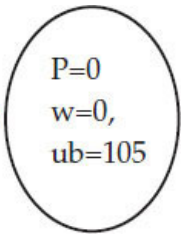
Here in B & B method we use fractional knapsack of greedy to solve find the upper bound and for the cost and weight of that node we use 0/1 knapsack method.

Suppose we are having 4 items as follows and we have to arrange them in knapsack of capacity 16.

Step 1: Now, as we arrange the items as in greedy algorithm according to descending order of their profit weight ratio i.e.

Step 2: Now, for the root node we have continuous solution- (1, 1, 1,1/3)

⇨ **UB** = 40+35+15+15=105



Step 3: Now as we have done in backtracking method we will add x1.

If we add x1 the value of UB remains same as in our continuous solution we have consider it.

⇒ Ub for node B remains same.

*When we not add x1 the value of continuous solution will be (0, 1, 1, 8//9). Ub for node C = 0+35+15+ (45*8/9)= 90*

Step 4: Since, the ub for node B is greater than that of node C. So we will consider node B Firstly. In this process again we will now consider item x2.

If we add x2 the value of UB remains same as in our continuous solution we have consider it.

⇒ Ub for node D remains same.

When we not add x2 the value of continuous solution will be (1, 0, 1, 8//9).

*⇒ Ub for node E = 40+0+15+ (45*8/9)= 95*

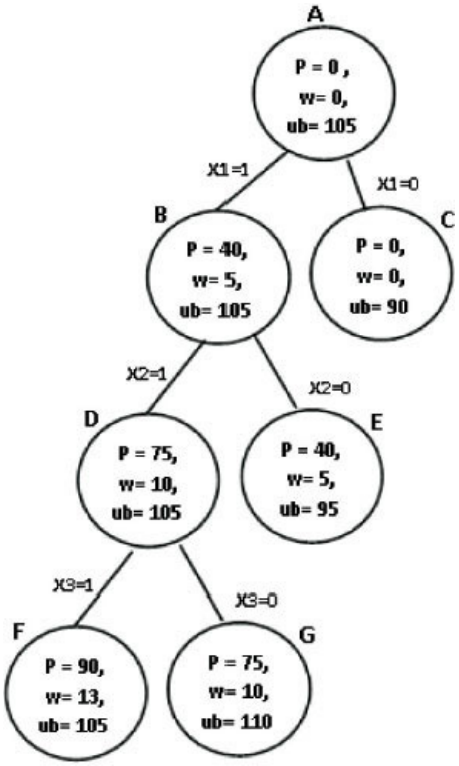
Step 5: Since the ub for node D is greater than that of node E. So we will consider node D Firstly, we will now consider item x3.

If we add x3 the value of UB remains same as in our continuous solution we have consider it.

⇒ Ub for node F remains same.

When we not add x3 the value of continuous solution will be (1, 1, 0, 6/9).

*⇒ Ub for node G = 40+35+0+ (45*6/9)= 110*



Step 6: Since the ub for node F and node G is almost equivalent. So we will consider both nodes. Firstly, for node F, we will now consider item x4.

If we add x4 the value of total weight will be greater than the capacity of knapsack.

⇒ This solution becomes infeasible.

When we not add x4 the value of continuous solution will be (1, 1, 1, 0).

⇒ Ub for node I = 40+35+15=90.

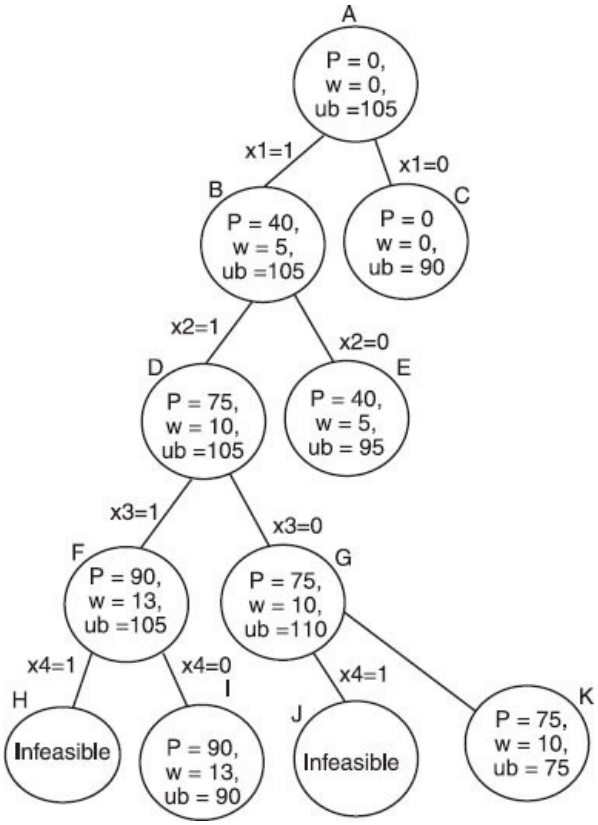
Now, for node G, we will now consider item x4.

If we add x4 the value of total weight will be greater than the capacity of knapsack.

⇒ This solution becomes infeasible.

When we not add x4 the value of continuous solution will be (1, 1, 0, 0).

⇒ Ub for node K = 40+35=75.



Step 7: Since the method is branch and bound and we are reducing here the number of searches. That's why step by step, we have avoid traversing the branches having Ub lesser than ub of nodes of same level. And when we got:

P= = ub, it becomes the final result for that particular branch. As we got in this example node I and K.

8.3 Travelling Salesman Problem

The traveling salesman problem comprises of a salesman and a set of cities, where he need to traversed at least once. The salesman has to visit every single one of the cities beginning from a specific one (e.g. the hometown) and coming back to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip.

A branch-and-bound algorithm comprises of the specification of all candidate solutions, where extensive subsets of inefficient competitors are discarded, by using upper and lower estimated bounds of the quantity being optimized.

The Branch and Bound strategy divides a problem to be solved into various sub- problems. It is a framework for taking care of a succession of sub problems each of which may have different conceivable solutions and where the solution decide for one sub- problem may affect the possible solutions of later sub-problems.

Assume it is required to constrain an objective function. Suppose that we have a methodology for getting a lower bound on the cost of any solution among those in the set of solutions represented by some subset. In the event the best solution found so far costs not as much as the lower bound for this subset, we require not exploring this subset by any means.

Let S be some subset of solutions.

Let L(S) = a lower bound on the cost of any solution having a place with S

Let C = cost of the best solution discovered up until this point

If C ≤ L(S), then there is no compelling reason to explore S since it doesn't contain any better solution.

If C > L(S), at that point we have to exploreS since it mighthave a better solution.

Lower Bound for a TSP:

Cost of the tour =

$$\frac{1}{2}\sum_{v \in V}$$

(sum of the osts of th two least cost edges adjacent to v)

Now,

The sum of the two tour edges adjacent to a given vertex v≥sum of the two edges of least cost adjacent to v Hence,

Cost of any tour =

$$\frac{1}{2}\sum_{v \in V}$$

(sum of the osts of th two least cost edges adjacent to v)

8.3.1 Applications

The TSP normally emerges as a sub problem in numerous transportation and logistics applications.

For example, arranging of a machine to bore gaps in a circuit board or other object.

In situation of holes to be drilled are the cities, and the cost of travel is the time it takes to move the drill head starting with one gap then onto the next.

Algorithm

1. *ALGORITHM TO CALCULATE BOUNDS*

FUNCTION CheckBounds(st,des,cost[n][n]) //Calculate the bounds

Global variable: cost[N] [N] - the cost assignment.

pencost[0] = t

for i← 0 to n -1

do

for j ← 0 to n-1

do

reduced [i][j] = cost [i][j]

end for

end for

for j ← 0 to n-1

do

reduced[st][j] = ∞

end for

for i←0 to n -1

do

reduced[i][des] =∞

end for

reduced[des][st] = ∞

RowReduction(reduced)

ColumnReduction(reduced)

pencost[des] = pencost[st] +row+col+cost[st][des]

return pencost[des]

end function

1. *Algorithm to find minimum value in row*

Function RowMinValue(cost[n][n],i) //Calculate min in the row

min = cost[i][0]

for j←0 to n -1

do

if cost[i][j] < min

then

min = cost[i][j]

end if

end for

return min

end function

1. *Algorithm to find minimum value in column*

Function ColMinValue(cost[n][n],i) //Calculate min in the col

min = cost[0][j]

for j←0 to n -1

do

if cost[i][j] < min

then

min = cost[i][j]

end if

end for

return min

end function

1. *Algorithm for row reduction*

Function Rowreduction(cost[n][n]) //makes row reduction

row = 0

for i←0 to n -1

do

rmin = rowmin(cost, i)

if rmin ≠ ∞

then row = row + rmin

end if

for j←0 to n -1

do

if cost [i][j] ≠ ∞

then

cost[i][j] = cost[i][j] - rmin

end if

end for

end for

end function

1. *Algorithm for column reduction*

Function Colreduction(cost[n][n]) //makes column reduction

col = 0

for j←0 to n -1

do

cmin = ColMinVal(cost, j)

if cmin ≠ ∞

then col = col + cmin

end if

for i←0 to n -1

do

if cost [i][j] ≠ ∞

then

cost[i][j] = cost[i][j] - cmin

end if

end for

end for

end function

1. *Main function*

Function Main //main function

for i←0 to n-1

do

select[i] = 0

end for

rowreduction(cost)

columnreduction(cost)

t = row + col

while allvisited(select) ≠ 1

do for i ← 1 to n-1

do

if select[i] = 0

then edgecost[i] = checkbounds(k, i, cost)

end if

end for

min = ∞

for i ← 1 to n-1

do

if select[i] = 0

then

if edgecost[i] < min

then

min = edgecost[i]

k =i

end if

end if

end for

select[k] = 1

for p←1to n-1

do

cost[j][p] = ∞

end for

for p←1 to n-1

do

cost[p][k] =∞

end for

cost[k][j] = ∞

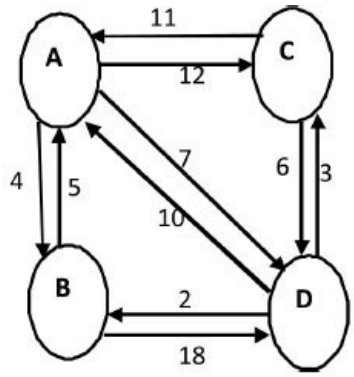
rowreduction(cost)

columnreduction(cost)

end while

end function

Example



The distance matrix with respect to this graph is: $\begin{bmatrix} \infty & 4 & 12 & 7 \\ 5 & \infty & \infty & 18 \\ 11 & \infty & \infty & 6 \\ 10 & 2 & 3 & \infty \end{bmatrix}$

Suppose we are taking A as a source node hence the tour must start from node A as well as also ends on node A. Now to compute distance by taking cost as a primary function, we will follow the following steps.

Step 1: The very first step is to reduce each and every row and column in such a way that each and every row and column must contain at least one zero value.

⇒ Reduce row 1 by 4, row 2 by 5 and so on..

Total cost for node1 = cost 1= reduction in row + reduction in column = 4+5+6+2+1=18

Step 2: After finding cost for node 1. Now we may move to node B, node C or it may be node D, depending on which node have minimum cost to reach from node A.

So, in process of this we will first consider the cost of node B into consideration i.e. compute cost from A -> B we will take above computed matrix

$$\begin{bmatrix} \infty & 0 & 7 & 3 \\ 0 & \infty & \infty & 13 \\ 5 & \infty & \infty & 0 \\ 8 & 0 & 0 & \infty \end{bmatrix}$$

a) In above resultant matrix M[A,B]=0

b) We set row A to ∞ and column B to ∞, matrix is $= \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 13 \\ 5 & \infty & \infty & 0 \\ 8 & \infty & 0 & \infty \end{bmatrix}$

c) Set M[B,A]=∞, matrix $= \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 13 \\ 5 & \infty & \infty & 0 \\ 8 & \infty & 0 & \infty \end{bmatrix}$

d) Now reduce the value of each and every row and column in such a way that each and every row and column must contain at least one zero value.

Total cost for node2 = cost 2= Cost 1+ reduction in row + reduction in column + M[A,B]

= 18+13+5+0=36

Again we will consider node C into consideration i.e. compute cost from A -> C we will take above computed matrix $\begin{bmatrix} \infty & 0 & 7 & 3 \\ 0 & \infty & \infty & 13 \\ 5 & \infty & \infty & 0 \\ 8 & 0 & 0 & \infty \end{bmatrix}$

a) In above resultant matrix M[A,C]=7

b) We set row A to ∞ and column C to ∞, matrix $= \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 13 \\ 5 & \infty & \infty & 0 \\ 8 & 0 & \infty & \infty \end{bmatrix}$

c) Set M[C,A]=∞, matrix is $= \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 13 \\ \infty & \infty & \infty & 0 \\ 8 & 0 & \infty & \infty \end{bmatrix}$

d) Now reduce the value of each and every row and column such that each and every row and column must contain at least one zero value. Since rows and columns are already reduced.

e) Total cost for node3 = cost 3= Cost 1+ reduction in row + reduction in column + M[A,C]

= 18+0 + 7=25

Next, we will consider node D into consideration i.e. compute cost from A -> D we will take above computed matrix $\begin{bmatrix} \infty & 0 & 7 & 3 \\ 0 & \infty & \infty & 13 \\ 5 & \infty & \infty & 0 \\ 8 & 0 & 0 & \infty \end{bmatrix}$

a) In above resultant matrix M[A,D]=3

b) We set row A to ∞ and column D to ∞, matrix is $= \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ 5 & \infty & \infty & \infty \\ 8 & 0 & 0 & \infty \end{bmatrix}$

c) Set M[D,A]=∞, matrix is $= \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ 5 & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty \end{bmatrix}$

d) Now reduce each and every row and column in such a way that each and every row and column must contain at least one zero value.

e) Total cost for node4 = cost 4= Cost 1+ reduction in row + reduction in column + M[A,D]

= 18+ 5+ 3 =26

Since the least cost covered in above three options is path from A->C Hence the next node after node A is C

Step 3: After finding cost for node 4. Now we may move to node B, or it may be node D, depending on which node has minimum cost from node C to reach.

So, in process of this we will first take node B into consideration i.e. compute cost from A -> C -> B we will take above computed matrix

$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 13 \\ \infty & \infty & \infty & 0 \\ 8 & 0 & 0 & \infty \end{bmatrix}$

a) In above resultant matrix M[C,B]=∞

b) We set row C to ∞ and column B to ∞, matrix will be $= \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 13 \\ \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & \infty \end{bmatrix}$

c) Set M[B,A]=∞, matrix is $= \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 13 \\ \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & \infty \end{bmatrix}$

d) Now reduce each and every row and column in such a way that each and every row and column must contain at least one zero value.

e) Total cost for node5 = cost 5= Cost 3 + reduction in row + reduction in column + M[C,B]

= 25+21+ ∞ = ∞

So, in process of this we will first take node D into consideration i.e. compute cost from A -> C -> D we will take above computed matrix

$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 13 \\ \infty & \infty & \infty & 0 \\ 8 & 0 & 0 & \infty \end{bmatrix}$

a) In above resultant matrix M[C, D]=0

b) We set row C to ∞ and column D to ∞,

c) Set M[D,A]=∞, matrix is $= \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty \end{bmatrix}$

d) No need of reduction => cost 6 = cost 3+ M[C, D] = 25

Step 4: Now only one node B left but we have to compute the cost for it. Compute cost from A -> C -> D -> B we will take above computed matrix

$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty \end{bmatrix}$

a) In above resultant matrix M[D,B]=0

b) We set row D to ∞ and column B to ∞,

c) Set M[B,A]=∞, matrix is $= \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ 5 & \infty & \infty & \infty \\ 8 & 0 & 0 & \infty \end{bmatrix}$

d) No need of reduction => cost 7 = cost 6+ M[D,B] = 25+ 0 = 25

REVIEW EXERCISE

Solve the knapsack problem for capacity = 16. For the given data

Items	W	p	Pi/wi
X1	3	45	15
X2	5	30	6
X3	9	45	5
X4	5	10	2

Find the solution for travelling salesman problem for the distance vector given by $\begin{matrix} \circ & 3 & 6 \\ 5 & \circ & 2 \\ 6 & \circ & \circ \end{matrix}$

MULTIPLE CHOICE QUESTIONS & ANSWERS

When the problem involves the allotment of n different facilities to n different tasks, it is often termed as an ……

(a) Transportation problem

(b) Game theory

(c) Integer programming problem

(d) Assignment problem

Ans. (d) Assignment problem

When there is no column and no row without assignment. In such case, the current assignment is……..

(a) Maximum

(b) Optimal

(c) Minimum

(d) Zero

Ans. (b) Optimal

Not traverse to the node already passed through or passing through every node at least once and only once. Such type of problem are called as

(a) Allocation problem

(b) Travelling problem

(c) Integer programming problem

(d) Routing problem

Ans. (d) Routing problem

An assignment problem can be formulated as a linear programming problem; it is solved by special method known as

(a) Gomers method

(b) Hungarian method

(c) Branch and bound method

(d) Queuing model

Ans. (b) Hungarian method

An airport service which parks all its limos at the airport can minimize its cost by using a proper sequence of order to pick up passengers from their houses and return to the airport using which algorithm:

(a) set covering problem

(b) traveling salesman problem

(c) knapsack problem

(d) fixed charge problem

Ans. (b) traveling salesman problem

A parlor lady carrying her tote containing makeup materials, can maximize her profit from one trip to the rural areas if she follows the strategy of loading her bag (with the "right" materials having maximum profitability per unit volume) by using

(a) set covering problem

(b) traveling salesman problem

(c) knapsack problem

(d) fixed charge problem

Ans. (c) knapsack problem

Branching in the branch and bound method refers to:

(a) adding a constraint

(b) removing a constraint

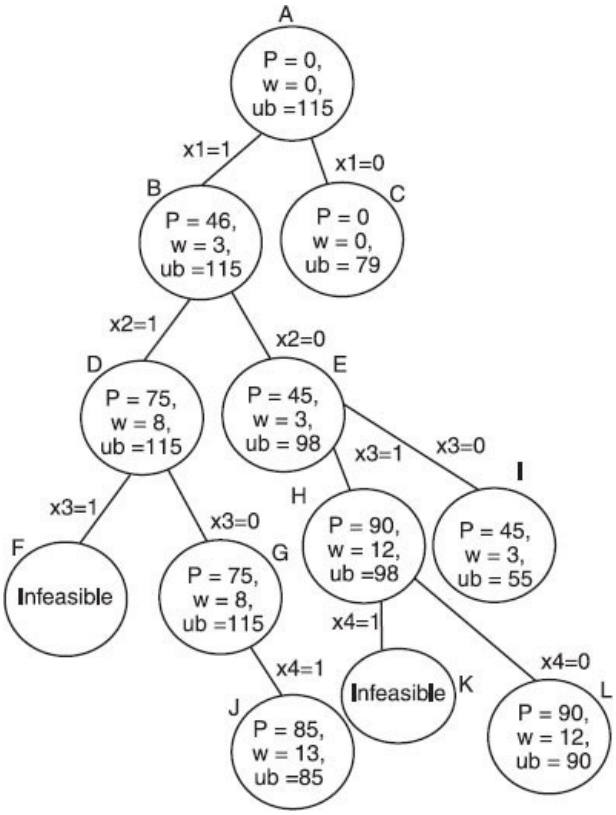
(c) either adding or removing a constraint

(d) removing a constraint and extending the feasible set

Ans. (a) adding a constraint

SOLUTION OF REVIEW EXERCISE

Ans. 1:



C HAPTER -9

String-Matching Algorithms

In this chapter student will understand:

What is string matching or pattern matching? From where this term came? Why one need to study this? What is the role of string matching in current world?

9.1 String Matching

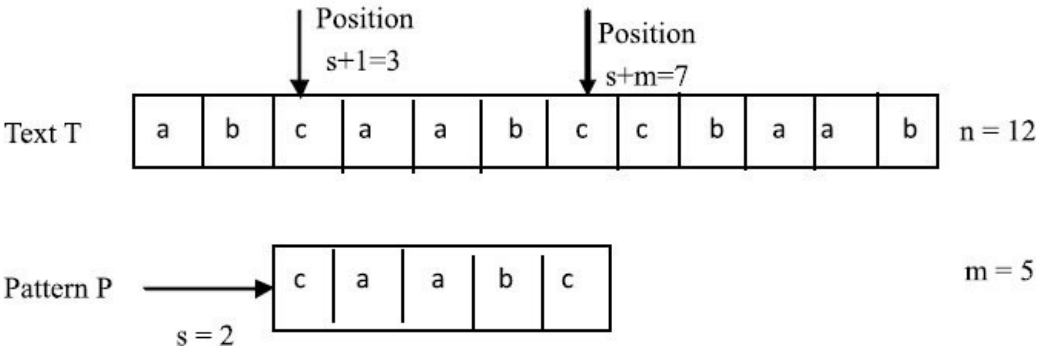
String matching or we may say pattern recognition is a problem for searching an object, called pattern , to be searched within a large object i.e. text under certain conditions and find out all occurrences of it.

In principle, pattern and text both will be in the form of an array of characters drawn from a finite alphabet, ∑. We may denote pattern by P [1…m] and text by T [1…n] where m and n are their respective length such that $n \geq m \geq 1$. If pattern P occurs in text T after s shifts then P[1…m] = T[s+1…s+m] where $n-m \geq s \geq 0$. If P occurs after finite shift s in T, then we may say that s is a valid shift otherwise it will be an invalid shift.

Algorithm: A Simple-Pattern-Finding algorithm can be considered as follows, whereas an input we have pattern P[1…m] of length m and text T[1… n] of length n such that $1 \leq m \leq n$. We need to list all of the possible occurrences of s, such that P occurs with shift s in T.

1. for s← 0 to n − m
    a. if (P[1… m] = = T[s + 1… s + m])
        i. {Output s}

Example:



The pattern occurs only once in the text after shift s = 2.

9.2 Notations and Terminology

*Let ∑ * (read "sigma-star") denotes the set of all finite-length strings formed using characters from the alphabet ∑. The zero-length empty string, denoted by | x |, also belongs to ∑*. The length of a string x is denoted by | x |. The concatenation of two strings x and y, where characters of x are followed by y which is denoted by xy , has length |x| + |y|.*

A string w is a prefix of x denoted by w x if x = wy , where y belongs to ∑. For example ab is a prefix of the string abccba and we may write it as ab ⊂ abccba. Again, a string w is a suffix of x denoted by w ⊂ x if x = yw where y belongs to ∑. In same example ba is a suffix of the string abccba and we may write it as ba ⊃ abccba.

Example: Again we may clear it with one more example:

Let S= attgtcg . Then the following are all prefixes of S :

a

at

att

attg

attgt

attgtc

attgtcg

Let S=attgtcg. Then the following are all suffixes of S :

g

cg

tcg

gtcg

tgtcg

ttgtcg

attgtcg

Now if we are illustrating this in a pictorial form:

Note: The empty string ε is a prefix and suffix to every string. Both ⊃ and ⊂ are ordered relation i.e. they are reflexive and transitive.

9.3 Lemma: (Overlapping and Suffix Lemma)

Suppose that x, y and z are three strings such that x ⊃ z and y ⊃ z. Then,

If |x|≤ |y|, then x ⊃ y.

If |y|≤|x|, then y ⊃ x.

If |x| = |y|, then x = y.

Proof: In first case we have x ⊃ z and y ⊃ z.

For x ⊃ z we must have a string w 1 such that z = w 1 x. Again, similarly for y ⊃ z we have w2 such that z = w 2 y.

So, z = w 1 x = w 2 y.

As we have |x|≤|y|, therefore w 2 must be a substring of w 1 and there must be a string w 3 for which

y = w 3 x.

Hence, x ⊃ y.

Similarly, we may prove rest of the two cases.

Example:

Suppose we have a string z = attgctg, x = tg and y = gctg which means |x|≤|y|.

Further, z = w 1 x where w 1 = attgc and z = w 2 y where w 2 = att.

As we know that, z = w 1 x = w 2 y and |x|≤|y|

i.e., attgctg = attgctg

Now, we may see here that there is some w 3 = gc for which y = w 3 x.

Hence it is proved that x ⊃ y.

9.4 Classification Of String Matching Algorithm

There are four types of algorithms used for string matching:

Naive String-matching algorithm

Rabin-Karp algorithm

String-Matching with automata

Knuth-Morris-Pratt Algorithm

9.4.1 Naive String Matching Algorithm

The Naive string matching algorithm results into all the possible occurrences of pattern P in text T by checking it (n - m + 1) times for all shifts.

Algorithm: For pattern P[1…m] of length m and text T[1… n] of length n such that 1 ≤ m ≤ n, we need to list all occurrences of s, such that P occurs with shift s in T.

n← length |T|

m← length |P|

for s← 0 to n - m

a. i← 1

while (P[i] = = T[i+ s] && i ≤ m)

a. i← i + 1,

if (i = = m + 1)

then print “P occurs at position s”

Example:

For T = abcdabbcbabb and P = abb, we may have n = 12 and m =3

Now, for loop will run from s ← 0 to 9.

For s = 0, the loop starts from the very first character of T and will match the successive three characters with pattern. If they match then it will show the occurrence, otherwise no occurrence is displayed and now go for next s i.e. for s = 1 and do the same process. This will continue till we reach at s = 9.

Similarly, till s= 4 we get all invalid shift.

Here P[1..3] = = T [1+ 4…3+4], hence s=4 is a valid shift. Same valid condition will occur at s=9.

Similarly, till s=4 we get all invalid shift

Here, p[1…3]==T{1+4…3+4], hence s=4 is a valid shift. Same valid condition will occur at s=9

Thus it is clear from the above given example that Naive String Matching Algorithm works as a sliding of pattern through the text which match all possible exact copy in the text if it finds same copy then it is called a valid shift otherwise it's an invalid one. The sliding process will continue from the start character of the text till the end character.

Analysis:

For a Text of length n← length |T| and pattern m← length |P,| the complexity is O((n-m +1) m .*

Best case analysis:

The algorithm will complete in minimum time iff :

Case: 1 The 1st character of pattern P is not present in the text T.

For example, T = abbacgde and P = fabb . Here the inner loop will execute only once for all shifts. Hence, T (n)=O(n) .

Case: 2 If pattern is in the first shift then T (n)=O(m).

Worst case analysis:

The algorithm will take maximum time to complete iff :

Case: 1 All characters of pattern are same as all characters of the text.

E.g. T = aaaaaaaaaaaaa and P=aaa

Case: 2 Only the last character is different

*E.g. T = aaaaaaaaaaaaa and P=aaab . Here in this case inner loop will execute from i =0→ m for all shifts. Hence T (n)=O(m *(n-m+1))*

9.4.2 Rabin Karp String Matching Algorithm

Though other algorithms have better worst case complexity, Rabin Karp algorithm has its own significance as it is easily generalized to all problems. It is mainly used in the field of 2-D pattern, fingerprint matching, DNA matching etc.

This algorithm uses hash values of pattern and 'm' characters subsequence of text for comparison. The hash value of a string is a numeric value. This numeric value is calculated by modular arithmetic, to make sure that this hash value can be stored in a word memory space. If the hash values matches, then algorithm will compare pattern and the 'm' character subsequence otherwise it will compare for hash values of next 'm' character subsequence of text. To do rehashing we need to take off the most significant digit and add the new least significant digit for in hash value.

Algorithm: For pattern P[1…m] of length m and text T[1… n] of length n such that 1 ≤ m ≤ n, we need to list all occurrences of s, such that P occurs with shift s in T.

1. n←length |T|
2. m←length |P|
3. h←d^{m-1} mod q for each character P[i] as a non-negative number < d, where d is the size of alphabet i.e. |Σ|.
4. p←0
5. t_0←0
6. for i←1 to m ► preprocessing
   a. p←(d p + P[i]) mod q //where q is a prime number, chosen such that all computation can be done with precision
   b. t_0 ← (d t_0 + T[i]) mod q
7. for s ← 0 to n – m← matching
   a. if p = = t_s then
      a. if P[1…m] = = T[s+1… s+m] then
      b. print " pattern occurs with shift s"
8. if s < n-m then
   a. t_{s+1} ←$(d(t_s-T[s+1]h)+T[s+m+t])$ mod q ► rehashing

Analysis : The complexity of preprocessing function is of order of p i.e. p is calculated Θ(n) and that of matching function or calculating t s is Θ((n-m +1) m).

Best case and average case analysis : If the expected number of valid shifts is small i.e. O(1) and the prime number is quite large then the complexity will be O(n + m) plus time taken to tackle spurious hits.

Worst case analysis: It is always possible to construct a scenario with a worst case complexity of O(nm). This, however, is likely to happen only if the prime number used for hashing is small.

Note 1: Here for computing hash values of all strings of length m i.e. 'p' in equal time we use **Horner's rule**.

The quadratic equation of polynomials is

$y = c_1 x^2 + c_2 x + c_3$

can be rearrange as $y = (c_1 x + c_2) x + c_3$

The cubic equation $y = c_1x^3 + c_2 x^2 + c_3 x + c_4$

can be arranged as $y = ((c_1x + c_2) x + c_3) x + c_4$

This arrangement is called Horner's rule by using it we calculated here value of 'p' and 't_s'.

$$p = \sum_{t=1}^{m} p[i].\ 10^{m-i}$$

$= P[m] + P[m-1] . 10^1 + P[m-2]. 10^2 + \ldots\ldots\ldots + P[1] . 10^{m-1}$

$= P[m] + 10. (P[m-1] + 10. (P[m-2] + \ldots\ldots + 10 P[2] + P[1])) \ldots .)$

Similarly we can compute $t_{s+1} = 10 (t_s - 10^{m-1} T [s+ 1]) + T [s+m+1]$

You may be more clear by the example of a text T = 12345234 and m = 5

Then for first shift we have $t_1 = 12345$

and for second shift rather than computing hash value of 23452 we have to compute value in form of:

$$\begin{aligned} 23452 &= 10 * (12345 - 10^{5-1} * T[0+1]) + T[0+5+1] \\ &= 10 * (12345 - 10000 * 1) + 2 \\ &= 10 * (12345 - 10000) + 2 \end{aligned}$$

This shows that each shift takes equal step of computation, hence will take constant time in execution of each shift.

Note 2: In matching process we use the *theory of Modular Equivalence* which states that two numbers are equivalent if they have same remainder after divided by same number i.e.

If a mod q == b mod q

Then we may say $a \equiv b$.

Example 1:

For T = 23590231415267399 and P = 31415, we may have n = 17 and m = 5

Now, d = 10 and q = 13

$\Rightarrow h = 10^{5-1} \bmod 13 = 3$

$p \leftarrow 0$

$t_0 \leftarrow 0$

For loop will run from i ←1 to 5.

For i = 1, p =(10 * 0 + 3) mod 13 = 3 mod 13 = 3 and
t_0 = (10 * 0+ 2) mod 13 = 2 mod 13 = 2

For i = 2, p =(10 * 3 + 1) mod 13 = 31 mod 13 = 5 and
t_0 = (10 * 2 + 3) mod 13= 23 mod 13 = 10

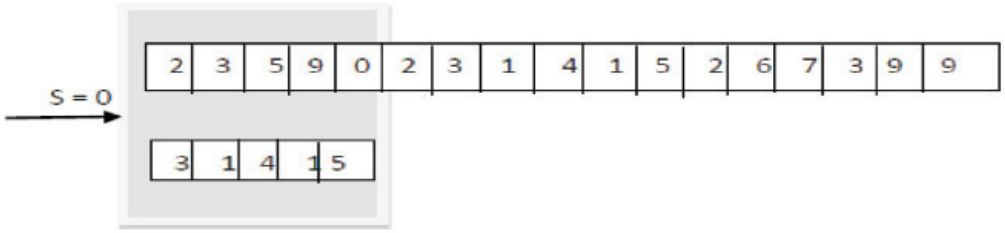
For i = 3, p=(10 * 5 + 4) mod 13 = 54 mod 13 = 2 and
t_0 = (10 * 10+ 5) mod 13 = 105 mod 13 = 1

For i = 4, p =(10 * 2 + 1) mod 13 = 21 mod 13 = 8 and
t_0 = (10 * 1+ 9) mod 13 = 19 mod 13 = 6

For i = 5, p =(10 * 8 + 5) mod 13 = 85 mod 13 = 7 and
t_0 = (10 * 6+ 0) mod 13 = 60 mod 13 = 8

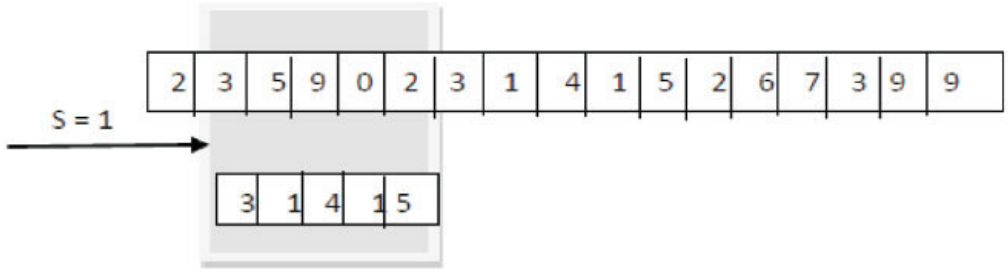
for s ← 0 to 17- 5 = 12

for the s=0, from very first character of T and will match the hash value i.e. t 0 of successive five characters with hash value of pattern i.e. p. If they match then it will match T[s…s+5] and P. If they match then it will show the occurrence, otherwise no occurrence is displayed and now go for next s i.e. for s=1 and do the same process. This will continue till we reach at s=12.



p = 7 and t_0 = 8
and s < 17 – 5 = 12
then $t_{0+1} = t_1$ = (10 (8 – 3* 3) + 2) mod 13 = (10 *2 + 2) mod 13
= 22 mod 13 = 9

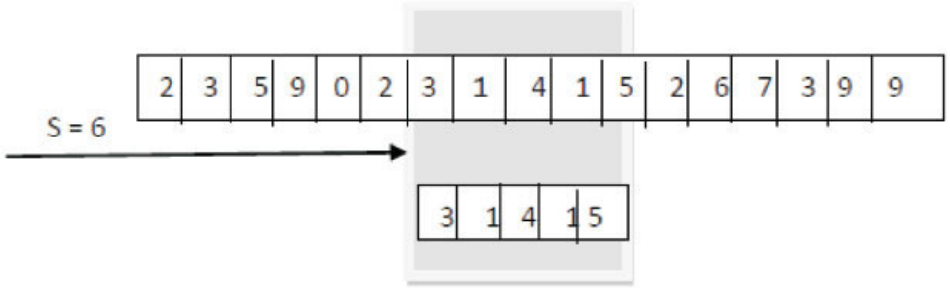
Hence s=0 is an invalid shift as for this pattern T[1.5]!=P[5] i.e. 23590!=31415



p = 7 and t_1 = 9
and s < 17 – 5 = 12
then $t_{1+1} = t_2$ = (10 (9 – 3* 5) + 3) mod 13 = (10 *–6 + 3) mod 13
= -57 mod 13 = 5

Hence s=1 is an invalid shift as for this pattern T[1.6]!=P[5] i.e. 35902!=31415

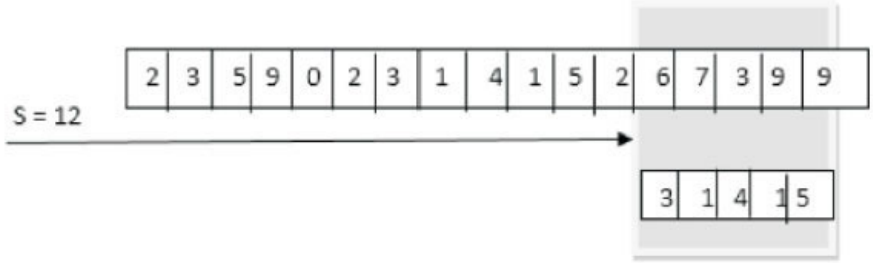
Similarly, till s= 5 we get all invalid shift.



p = 7 and t_6 = 7 an d T[6..11] != P[5] i.e. 31415 != 31415.

Hence s=6 is an invalid shift as for this pattern

Again we will get all invalid shifts till s = 11.



Here, we again get p = 7 and t_{12} = 7 but T[13…19] != P[5]. So this is a spurious hit.

Note: if p = ts but P [1..m] != T[s+1..s+m] then it is called a spurious hit.

Example 2:
For T = *abcdabbcbabb* and P = *dabb*, we may have n = 12 and m = 4
Now, d = $|\Sigma|$= 5 and q = 13
$\Rightarrow$ h = 5^{4-1} mod 13 = 8
p $\leftarrow$ 0
$t_0 \leftarrow 0$
For loop will run from i $\leftarrow$ 1 to 4.
For i = 1, p =(5 * 0 + ascii(d)) mod 13 = (5 * 0 + 100) mod 13 = 100 mod 13 = 9 and
t_0 = (5 * 0+ ascii(a)) mod 13 = (5 * 0+ 97) mod 13 = 97 mod 13
= 6
For i = 2, p = (5 * 9 + 97) mod 13 = 12 and
t_0 = (5 * 6 + 98) mod 13= 11

For i = 3, p =(5 * 12 + 98) mod 13 = 2 and
t_0 = (5 * 11+ 99) mod 13 = 11
For i = 4, p =(5 * 2 + 98) mod 13 = 4 and
t_0 = (5 *11+ 100) mod 13 = 12
for s ← 0 to 12 – 4 = 8
for the s = 0, p = 4 and t0 = 12
Hence t 1 = (5 (12 – ascii(T[1]) 5) + ascii(T [5])) mod 13
= (102 (12 – 98*5) + 98) mod 13 = 5
For s = 1, p=4 and t1 = 5
Hence t2 = (5 (5 – 99*8) + 98) mod 13
= 2
For s= 2, p = 4 and t2 = 2
Hence t3 = (5 (2 – 100*8) + 99) mod 13 = 4
Here t3 = = p = 4

Hence we got valid shift at s=3 because here we got T[3…6]==P[4]

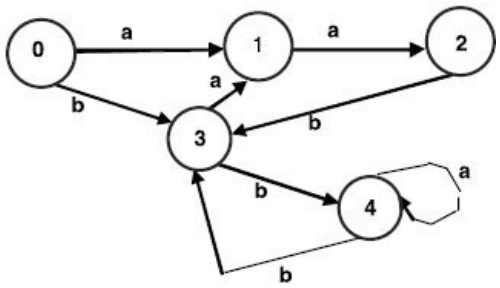
You can similarly solve it further to find more valid, invalid and spurious hit.

9.4.3 String-Matching with Automata

Creating automata is a way to improve the run time of string matching algorithms which has knowledge of what we should match next.

A finite automaton accepts strings in a particular language. It starts from state q0 and read one character at a time from input string. It makes transitions F based on this input string and if reaches at the end then it must be one of the accepted states by that automata.

A finite automata is consist of a tuple of (Q, q_0, A, Σ, δ) where
Q is a finite set of states
$q_0 \in Q$, is the initial state
$A \subseteq Q$, is the set of all accepted states.
Σ is input alphabet set
$\delta : Q \times \Sigma \rightarrow Q$ is the transition function.
For Example:
Q = { 0,1,2,3.4}
q_0 = 0
A = { 2,3}
Σ = {a, b }
δ(q, σ) =



q \ σ	a	b
0	1	3
1	2	3
2	2	4
3	1	4
4	4	3

Table: 9.1

For a given input string ' x ' from alphabet set '∑', a finite automata will

Start from q 0

Read character one by one from the string 'x' and change the state according to the δ(q, σ)

And move from state q to δ(q,a)

It will accept the string 'x' if it ends in a acceptable state i.e. if $q \in A$

and

Rejects if it ends in an unacceptable states i.e. $q \notin A$.

For example: For above given example if we have string x=bbbaabbaaab then

Character (σ)		b	b	b	a	a	b	b	a	a	a	b
New State(q)	0	3	4	3	1	2	3	4	4	4	4	3

Table: 9.2

As we see from the above given Table 9.2 that the string ends up in one of the acceptable states i.e. '3', hence this string is acceptable.

Now as you all become familiar with working of finite automata then we could easily move to its string matching concept.

A finite automaton induces a final-state function Φ

$-\phi: \Sigma^* \rightarrow Q$, such that $q=\phi(x)$ is the state M is in after scanning the string x
$-M$ accepts a string x if and only if $\phi(x) \in A$
–recursive definition of ϕ
$\phi(\epsilon) = q_0$
$\phi(xa) = \delta(\phi(x), a)$ for $x \in \Sigma^*, a \in \Sigma$

If we want to use automata for string matching we will need to build an automat on for each pattern P[1…m]. This has to be done as a preprocessor state. For this we have, the state set Q is 0,1,…,m, where start state q 0 is state 0 and state m is the only accepting state, the transition function is defined as δ (P, a)= = σ (P q a) for any state q and character a and Suffix function σ for the given pattern P [1…m] where

$-\sigma: \Sigma \{0,1,\ldots,m\}$ such that $\sigma(x) = \max \{k: P_k \sqsupset x\}$ is the length of the longest prefix of P that is a suffix of x

For Example : For pattern P = ababa in the above given figure

We have,

$\delta(q, a) = \delta(\phi(q), a) = \delta(P_q a)$ where a is notation for all possible characters in the given pattern.

Hence here a = {a,b}

⇒ for start state we have,

$\delta(0, a) = \delta(\phi(0), a) = \sigma(P_0 a)$

where $\delta(\phi(0), a) = 1$ as we know at state 0 after inputting 'a' we move to state 1and

and $\sigma(P_0 a) = \sigma(a) = 1$

Algorithm

To clarify the operation of a string-matching automaton, we now give a simple, efficient program in finding occurrences of a pattern P of length m in an input text T [1 - n]. As for any string-matching automaton for a pattern of length m , the state set Q is {0, 1,…, m }, the start state is 0, and the only accepting state is state m and transition function is δ(q,a)

FINITE-AUTOMATON-MATCHER (T, P, Σ, m)

n ← length [T]

δ ← compute Transition Function (P, ∑)

q ← 0

for i ← 1 to n

a. do q ← δ(q , T [i])

b. if q = m

then print "Pattern occurs with shift" i - m

COMPUTE-TRANSITION-FUNCTION (P, Σ)

1. $m \leftarrow length\,[P]$
2. for $q \leftarrow 0$ to m
3. do for each character $a \in \Sigma$
   a. do $k \leftarrow \min(m+1, q+2)$
   b. repeat $k \leftarrow k-1$
   c. until $P_k \sqsupset P_q a$
   d. $\delta(q,a) \leftarrow k$
4. return δ

Analysis: The running time of COMPUTE-TRANSITION-FUNCTION is O (m ³ |∑|), because the outer loops contribute a factor of (m |∑|) ,the inner repeat loop can run at most m +1 times, and the test P k ⊃ P g a can require comparing up to m characters. Once automata is computed the matcher takes Θ(n) run time complexity.

For example: For a given text T[i]= abababacbac and P= abababacba for which only acceptable state is '3'.

Firstly compute transition function δ

m=8 and ∑={a,b,c}

For q ← 0 to 8

A. For q = 0

i. For 'a'	ii. For 'b'	iii. For 'c'
do k = min(8+1, 0+2)	k = 2	k = 2
⇒ k = 2		
Repeat k = 2-1 = 1	k = 1	k = 1
Until $P_1 \sqsupset P_0$ a	until $P_1 \sqsupset P_0$ b	until $P_1 \sqsupset P_0$ c
⇒ a ⊐ a	⇒ a ! ⊐ b	⇒ a ! ⊐ c
δ (0, a) = 1		
	k = 0	k = 0
	$P_0 \sqsupset P_0$ b	$P_0 \sqsupset P_0$ c
	ε ⊐ b	δ⊐ c
	δ (0, b) = 0	δ (0, c) = 0

B. For q = 1

i. For 'a'	ii for 'b'	iii for 'c'
Do k = min (9, 3)		
⇒ k = 3		
Repeat k = 2	Repeat k = 2	Repeat k = 2
Until $P_2 \sqsupset P_1$ a	Until $P_2 \sqsupset P_1$ b	until $P_2 \sqsupset P_1$ c
b ! ⊐ a a	b ⊐ a b	b !⊐ a c
k = 1	δ (1, b) = 2	k = 1
$P_1 \sqsupset P_1$ a		$P_1 \sqsupset P_1$ c
a ⊐ a a		a !⊐ a c
δ (1, a) = 1		k = 0
		$P_0 \sqsupset P_1$ c
		ε ⊐ a c
		δ (1, c) = 0

C. for q = 2

i. for 'a'	ii. For 'b'	iii For 'c'
δ (2,a) = 3	δ (2, b) = 2	δ (2, c) = 0

D. for q = 3

i. for 'a'	ii. For 'b'	iii For 'c'
δ (3,a) = 3	δ (3, b) = 4	δ (3, c) = 0

E. for q = 4

i. for 'a'	ii. For 'b'	iii For 'c'
δ (4,a) = 5	δ (4, b) = 4	δ (4, c) = 0

F. for q = 5

i. for 'a'	ii. For 'b'	iii For 'c'
δ (5,a) = 5	δ (5, b) = 4	δ (5, c) = 6

G. for q = 6

i. for 'a'	ii. For 'b'	iii For 'c'
δ (6,a) = 5	δ (6, b) = 7	δ (6, c) = 6

H. for q = 7

i. for 'a'	ii. For 'b'	iii For 'c'
k = min (8+1, 7+2)		
⇒ k = 9		
δ (7,a) = 8	δ (7, b) = 7	δ (7, c) = 6

I. for q = 8

k = min (8+1, 8 +2)

⇒ k = 9

Which is similar to value for q = 7

Now calculate FINITE-AUTOMATON-MATCHER (*T*, P, ∑, *m*)

n ←11

δ ← we have already compute Transition Function (P, Σ)

q ← 0

for *i* ← 1 **to 11**

1. For i = 1
   do *q* ← δ(0, *T*[1])
   = δ (0, a) = 1
   if *q* = *m* i.e. 1 != 8
2. For i = 2
   do *q* ← δ(1, *T*[2])
   = δ (1, b) = 2
   if q = m i.e. 2 != 8
3. For i = 3
   do q ← δ (2 , T [3])
   = δ (2, a) = 3
   check q = m i.e. 3 ! = 8
4. For i = 4
   do *q* ← δ(3, *T*[4])
   = δ (3, b) = 4
   if *q* = *m* i.e. 4 != 8
5. For i = 5
   do *q* ← δ(4, *T*[5])
   = δ (4, a) = 5
   if *q* = *m* i.e. 5 != 8
6. For i = 6
   do q ← δ(5 , T [6])
   = δ (5, b) = 4
   check q = m i.e. 4 ! = 8
7. For i = 7
   do q ← δ(4, T[7])
   = δ(4, a) = 5
   if q = m i.e. 5 != 8
8. For i = 8
   do q ← δ(5, T[8])
   = δ(5, c) = 6
   if q = m i.e. 6 != 8
9. For i = 9
   do q ← δ(6, T[9])
   = δ (6, b) = 7
   if q = m i.e. 7 != 8
10. For i = 10
    do q ← δ(7, T[10])
    = δ (7,a) = 8
    if q = m i.e. 8 = = 8

then print "Pattern occurs with shift" *i – m i.e. 10 – 8 = 2 shifts.*

9.4.4 Knuth-Morris-Pratt Algorithm

In Naive (brute force) algorithm if a mismatch occurs at P[j] where j >1 then it slides only by one which is simply a wastage of information.

Example For T= abcdabbcbabb and P= abb, we may have n=12 and m=3

If we already know that there is a mismatch at position 3 then there is no need to slide by s=1 and we may start from shift to s=3.

In case of a mismatch or a match it uses the notion border of the string. KMP uses information gained from previous comparison. It computes a failure function (f) that indicates how many times last comparisons is refused in case of some mismatch.

For this, we calculate a prefix function $\pi[q]=\max\{k : k< q \ \& \ P_k \sqsupset P_q$ i.e.

π[q]=Length of the largest prefix of P q which is a proper suffix π q of create a table for this for matching purpose.

Algorithm

KMP Matcher(T,P)

1. $n \leftarrow length[T]$
   $m \leftarrow length[P]$
2. $\pi \leftarrow$ Compute prefix function(P)
3. $q \leftarrow 0$ //q represents number of characters matched
4. for $i \leftarrow$ 1 to n do
   i. while q > 0 & $P[q+1] \neq T[i]$
      a. do $q \leftarrow \pi[q]$ // next character doesn't match then // shift according to prefix function
   ii. if $P[q+1] == T[i]$
   iii. then $q \leftarrow q+1$ //to match next character
   iv. if $q == m$
   v. then print" pattern occur after shift i-m"
      $q = \pi[q]$ // to look for next occurrence in text

Compute prefix function (P)

1. $m \leftarrow length\ [P]$
   $\pi[1] \leftarrow 0$
2. $k \leftarrow 0$
3. for $q \leftarrow 2$ to m do
   i. while $k > 0$ & $P[k=1] \neq P[q]$
      1. do $k \leftarrow \pi[k]$
   ii. end while
   iii. if $P[k+1] == P[q]$ then
4. $k \leftarrow k+1$
5. end if
6. $\pi[q] \leftarrow k$

   return π

Analysis: Here in this algorithm prefix function has complexity of Θ(m) and a matcher has complexity of Θ(n).

Example: For a given pattern P= ababaca , and text T=abacababaca compute prefix function and match using KMP algorithm.

m=7

π[l]←0

k←0

for q=2 to 7

i. for q=2, k=0
neither while execute nor
if condition is true
⇒π[2]=0

ii. for q=3, k=0
if condition is true
⇒k=1
π[3]=1

iii. for q=4, k=1
if condition is true
⇒k=2
π[4]=2

iv. for q=5, *k*=2
if condition is true
⇒*k*=3
π[5]=3

v. for q=6, k=3
while will execute
k←π[3]=1
π[6]=1

vi. for q=7, k=1
while will execute
k←π[1]=0
π[7]=0

The tabular form of this prefix function will be

q	1	2	3	4	5	6	7
P	a	b	a	b	a	c	a
π	0	0	1	2	3	1	0

Table 9.3: Prefix function for pattern ababaca

Matcher(T,P)

n=11,q=0
for i=1 to 11

i. for i=1
if P[1]==T[1]
then q= 1

ii. for i=2
if P[2]==T[2]
then q=2

iii. for i=3
if P[3]==T[3]
q=3

iv. for i=4
while will execute
q=π[3]=1

v. for i=5
while will execute
q=π[7]=0
if P[1]==T[5]
q=1

vi. for i=6
if P[2]==T[6]
q=2

same step will follow till i=10

vii. for i=11
if P[7]= =T[11]
q = 7
if q == m

print pattern occur after shifts 4
q=π[7]=0

REVIEW EXERCISE

A finite automaton to match pattern 'ababc' over alphabet={a,b,c} and A=5. Matching pattern 'ababc' in text 'caabaabcababababccb'.

Check occurrence of pattern 'ababb' in text 'ababaabbababba' using Knuth Morris Pratt algorithm.

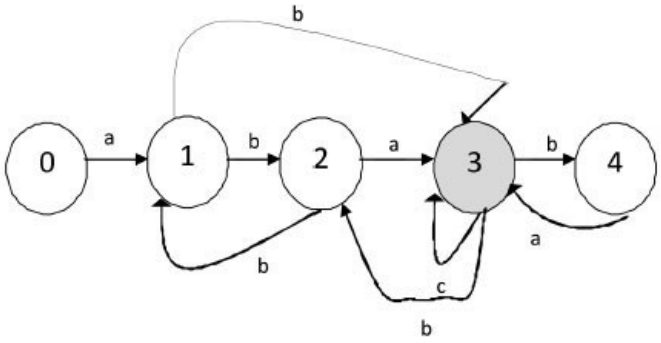
Find first match of pattern 'needle' in text 'hereisaneedleoutofmanyneedlesonthetable' using Naive String matching algorithm.

For modulo q=11, how many spurious hits does the Rabin-Karp matcher encounter in the text T=3141592653589793 when looking for the pattern P=26?

Given T = accgtgtaactctacgagacctacgaaccgt and the pattern P = tacgagacctacga , solve the string matching algorithm using Rabin Karp algorithm.

Given T=aaaaaaaaaa and P=aaaaa , what would be the difference in the step wise execution of Naive string matching algorithm and KMP algorithm?

Compute transition function using finite automata for a given text T[i]= abababacaba and P= ababaca.



Character (σ)		b	b	b	a	a	b	b	a	a	a	b
New state(q)	0	3	4	3	1	2	3	4	4	4	4	3

C HAPTER -10

P And NP Problems

In this chapter student will understand:

What is P problem, NP problem, NP hard and NP complete problem? How they are different with each other? What are their areas of application?

10.1 Introduction

Till now we have learned about efficient and successful algorithms for different problems, like knapsack problem, traveling salesman problem, etc.

But, computer can't solve ev ery single computational problem. Some of the computational problems can't be solved even by providing unlimited time!

For example,we can consider Turing Halting problem where the case is that if we give a program and an input, and do not bother whether it will terminate or continue with an infinite loop.

NP-complete problems are another story to be resolved; these are problems whose status is obscure. None of the polynomial time algorithms has been found for any NP- complete problem. Even reverse has not been proved that no any polynomial-time algorithm exists for any NP-complete problem. The interesting part is that if any of the NP-complete problems can be reduced in polynomial time, then each one of them can be handled.

A very famous open question in Computer Science: P=NP?

10.2 Polynomial (P) Problems

A problem is said to be polynomial if there exists an algorithm that can solve the problem in order of time T(n)=O(n c), where c is a constant.

Some Examples of polynomial problems:

Sorting: O(n log n)=O(n ²)

All-pairs shortest path: O(n ³)

Minimum spanning tree: O(E log E)=O(E ²)

For an exponential problem, an algorithm is developed which solves in exponential time such that time is represented by O(n u(n)), where u(n) goes to infinity as n goes to infinity. This problem can't be solved in polynomial time.

10.3 Nondeterministic Polynomial(NP) Problems

Definition 1 : A problem is said to be Non-deterministically Polynomial (NP) if we can find a non-deterministic Turing machine that can solve the problem in a polynomial number of non-deterministic moves.

For those who are not familiar with Turing machines, two alternative definitions of NP will be developed.

Definition 2: A problem is said to be NP if its solution originates from a limited arrangement of potential solutions, and it requires polynomial time to check the correctness of a candidate solution.

Remark: It is much easier and faster to "grade" a solution than to find a solution from scratch.

NP is used to define all type of non-deterministically solvable polynomial problems. Hence, P is a subset of NP.

Definition 3: There is one more way to define NP. We can present an imaginary, non-implementable instruction, which we call "choose()".

Behavior of "choose()": If a problem has a solution of N components, choose(i) will return the ith component of the CORRECT solution in steady time. If a problem has no solution, choose(i) which simply returns any garbage value.

A step to solve NP-Problem: An NP algorithm is an algorithm that has 2 stages:

The first stage is a guessing stage that uses function choose () to find a solution to the problem.

The second stage checks whether the solution of first stage is right or wrong. In this stage we have polynomial time for n input.

10.3.1 Generalize Algorithm For NP Algorithm

Begin / The following for-loop is the guessing stage*/*

for i ← 1 to N do

X[i] ← choose(i);

Endfor / Next is the verification stage */*

Write code without using "choose" and it checks below condition.

if X[1:N] is a right solution to the problem.

End

Definition 3: A problem is said to be NP if there exists an NP algorithm for it.

10.3.2 Example for an NP problem:

1. The Hamiltonian Cycle (HC) problem

   Input: Given graph G.

   Question: Does G have a Hamiltonian Cycle?

   Algorithm- Here is an NP algorithm for the HC problem:

   1. begin
      /* The following for-loop is the guessing stage*/
   2. for i ←1 to n do
   3. X[i] ←choose(i);
   4. endfor
      /* Next is the verification stage */
   5. for i ← 1 to n do
      a. for j ← i+1 to n do
         I. if X[i] == X[j] then
         II. return(no)
         III.endif
      b. endfor
   6. endfor
   7. for i=1 to n-1 do
      a. if (X[i],X[i+1]) is not an edge then
      b. return(no);
      c. endif
   8. endfor
   9. if (X[n],X[1]) is not an edge then
   10. return(no);
   11. endif
   12. return(yes);
   13. end

Analysis: The time complexity of HC is O(n), and the verification stage needs $O(n^2)$ time. So, HC is one of the NP problems.

1. *The K-clique problem*

Input: A graph G and an integer k

Question: Does G have a k-clique?

Algorithm

Here is an NP algorithm for the K-clique problem:

```
1. begin
/* The following for-loop is the guessing stage*/
2. for i ← 1 to k do
3. X[i] ← choose (i);
4. Endfor
   /* Next is the verification stage */
5. for i ← 1 to k do
   a. for j ← i+1 to k do
         I.   if (X[i] = X[j] or (X[i],X[j]) is not an edge) then
         II.  return(no);
         III. endif
   b. endfor
6. endfor
7. return(yes);
8. end
```

Analysis: The solution size of the k-clique is O(k)=O(n), and the time of the verification stage is O(n ²). Therefore, the k-clique problem is NP.

10.4 Optimization Problems and Decision Problems

NP-completeness has been studied in the framework of decision problems. Most problems are not decision problems, but optimization problems (where some value needs to be minimized or maximized). In order to apply the theory of NP-completeness to optimization problems, we must recast them as decision problems. Below is an example showing transformation of an optimization problem into a decision problem.

Example: Let's consider SHORTEST-PATH problem, given an undirected graph G =(V, E) where we have to find the shortest path. A SHORTEST-PATH instance contains a particular graph and two vertices of that graph. The solution for this must have a sequence of vertices in the graph, which may contain an empty sequence to denote that there no path exists. Thus the problem SHORTEST-PATH is a relation that associates each instance of a graph and two vertices with a solution (to be specific a shortest path in this case).

Note : It may be possible in some cases that a given instance may have no solution, or may have precisely one solution, or multiple solutions.

A decision problem related to the SHORTEST-PATH problem as discussed above is: For a given graph G=(V, E) , two vertices u, eV, and a non-negative integer k, does a path exist in G between u and v whose length is at most k?

Note : The decision problem PATH is one way of casting the original optimization problem as a decision problem. We have done this by imposing a bound on the value to be optimized. This is one of the ways of transforming an optimization problem into a decision problem.

Conventionally, easy optimization problem results into an easy related decision problem. Similarly, if it is justified that a decision problem is hard, then it is also proved that related optimization problem is also hard.

10.5 Reduction

Let L 1 and L 2 are two decision problems. Suppose Algorithm A 2 used to solve problem L 2 . That is, if y is an input for L 2 then algorithm A 2 will provide answer in terms of 'Yes' or 'No', depending upon whether y belongs to L 2 or not. The idea is to find a transformation from problem L 1 to L 2 so that the algorithm A 2 can be a part of an algorithm A 1 to find solution of L 1 .

Reduction refers to reducing an unsolved problem in form of the existing solved problems to save time. For example, consider a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. For this we can use the code for Dijkstra's algorithm to find shortest path and take log of all weights, apply on this code and find the minimum product path. By

doing this we don't need to write code for the current problem.

10.6 NP Hard Problems

Some problems can be translated into one another in such a way that the shortest solution to a problem would automatically give us an efficient solution to the other. There are some problems that every single problem in NP can be translated into, and a fast solution to such a problem would automatically give us a fast solution to every problem in NP. These groups of problems are known as NP-Hard .

Assume that there is a problem (H) which we believe is NP-complete (solution is verifiable in PTIME). Let there be an intermediate step (subroutine S) which must be solved in order to solve for H. If S is said to be NP-complete, then H is at least as hard as S. In fact H is harder than S. Thus H is harder than a NP-complete problem. So it needs to be in a complexity class of its own and hence H belongs to a class known as the NP-hard class. However note that H is not strictly NP-hard. This leads us to conclude that NP-complete is a class of problems that fall under both NP and NP-hard. Hardest of the NP problems is NP-complete and that of NP-complete is NP-hard.

10.6.1 Strictly NP-Hard Problem

NP-complete problems are optimization, search or decision problems.The classical TSP (Travelling Sales Person) is an optimization problem. For a problem to be strictly NP-hard it should be neither an optimization/decision/search problem.

10.6.2 NP-Complete Problems:

Some problems in NP-Hard group are not actually belong to NP group; the group of problems that belongs to both NP and NP-Hard is called NP-Complete .

Definition: A problem R is NP complete if and only if R is NP.

A P problem is NP complete if it is NP, means every NP problem must be reducible into P in polynomial time.

An equivalent but casual definition: A problem R is NP-complete if R is the “most difficult” of all NP problem set.

10.6.2.1 Theorem 1:

Let P and R be two problems. If P can be reduced to R and R is polynomial, then P is also polynomial problem.

Proof: Let T transforms P to R. T is a polynomial time algorithm that transforms I P to I R such that

Answer (Q P ,I P)=Answer(Q R ,I R)

Let A R be the polynomial time algorithm for problem R. Clearly, A takes as input I R , and returns as output Answer (Q R ,I R)

Design a new algorithm A P as follows:

Algorithm

A P (input: I P)

begin

I R ← T(I P);

x ← A R (I R);

return x;

end

Note:

The algorithm A P returns the correct answer (Q P ,I P) because,

x=A R (I R)=Answer(Q R ,I R)=Answer(Q P ,I P)

The algorithm A P takes polynomial time because both T and A R are reducible according to theorem 10.6.2.1.

10.6.2.2 Theorem 2

A problem R is NP-complete if R is NP, and there exists an NP-complete problem R0 that reduces to R.

Proof: Since R is NP, it remains to show that any arbitrary NP problem P reduces to R.

Let P be an arbitrary NP problem.

Since R 0 is NP-complete, it follows that P reduces to R 0 and since R 0 reduces to R, it follows that P reduces to R (by transitivity of transforms).

The previous theorem accounts a strategy for proving new problems to be NP complete. Specifically, to prove a new problem R to be NP-complete, the following steps are sufficient:

Prove R to be NP

Find an already known NP-complete problem R0, and come up with a transform that reduces R 0 to R.

We need at least one NP-complete problem to implement this strategy. Such problems are provided by Cook's Theorem below.

10.7 Cook's Theorem

Theorem: an NP-complete problem exists.

For example, The Satisfiability (SAT) problem is N P-complete.

Definition: We may say that a problem A in NP is NP-complete when, for every other problem B in NP, we have B < A.

According to this definition, by using the notion of reduction, if B < A, then the two problems are very closely related; for instance Hamiltonian cycle and longest path are both about finding very similar structures in graphs.

We prove this by example. One NP-complete problem can be found by modifying the halting problem (which without modification is un-decidable).

10.8 Bounded Halting

This problem takes as input a program H and a number N. The problem is to find data which, when given as input to H, causes it to stop in at most N steps.

B(H, N): Bounded halting function

More clearly, if we have a text with size H and its execution terminates after N time units then the function solving this is computable and is considered as a bounded halting problem.

Bounded halting is NP-complete as it is reducible to an NP-complete problem. We can justify this!

Let us assume that there exist a problem PP and it is reducible in NP. Thus we can say that there can be any program XYZ which tests the solution to PP and terminates in polynomial time p(n), with proper results. Using this we can further write another program XYZ' which enters into an infinite loop resulting into termination without proper results.

Using above scenario, if we solve bounded halting, we can also solve PP by passing XYZ' and p(n) as parameters to B() like B(XYZ', p(n)).

Hence, bounded halting is NP-complete.

REVIEW EXERCISE

Define P problem, NP Problem and NP hard Problem?

Explain Cook's theorem with an example.

What is bounding halt problem?

Differentiate between P problem, NP Problem, NP complete problem and NP hard Problem with the help of Venn diagram.

MULTIPLE CHOICE QUESTIONS & ANSWERS

Suppose there is a polynomial time algorithm that correctly computes the largest clique problem for a given graph. In this scenario, which one of the following Venn diagram represents the correct complexity for the classes P, NP and NP Complete (NPC)?

(a) A

(b) B

(c) C

(d) D [GATE 2014]

Ans. (d)

Explanation: Largest Clique is an NP complete problem. If one NP complete problem can be solved in polynomial time, then all of them can be. So, NPC set becomes equals to P.

Assuming P!=NP, which one of the following statements is true?

(a) NP-Complete=NP

(b) NP-Complete ∩P=Φ

(c) NP-Hard=NP

(d) P=NP-Complete

Ans. (b) NP-Complete ∩ P=Φ

Explanation: The answer is B (no NP-Complete problem can be solved in polynomial time). If one NP-Complete problem can be solved in polynomial time, then all NP problems can be transformed to solve in polynomial time. If this will be the case, then NP and P set become same which contradicts the actual condition.

Let S be an NP-complete problem and Q and R be two other problems not known to be in NP. Q is polynomial time reducible to S and S is polynomial-time reducible to R. Which one of the following statements is true? (GATE CS 2006)

(a) R is NP-Complete

(b) R is NP-hard

(c) Q is NP-Complete

(d) Q is NP-hard

Ans. (b) R is NP-hard

Explanation:

(a) Incorrect because R is not in NP. A NP-Complete problem has to be in both NP and NP-hard.

(b) Correct because a NP Complete problem S is polynomial time reducable to R.

(c) Incorrect because Q is not in NP.

(d) Incorrect because there is no NP-complete problem that is polynomial time Turing-reducible to Q.

The problem 3-SAT and 2-SAT are

(a) both in P

(b) both NP-Complete

(c) NP-Complete and in P respectively

(d) un-decidable and NP-complete respectively

Ans. (c) NP-Complete and in P respectively

Explanation: The Boolean satisfiability problem (SAT) is a decision problem, whose instance is a Boolean expression written using only AND, OR, NOT, variables, and parentheses. The problem is: given the expression, is there some assignment of TRUE and FALSE values to the variables that will make the entire expression true. A formula of propositional logic is said to be satisfiable if logical values can be assigned to its variables in a way that makes the formula true.

3-SAT and 2-SAT are special cases of k-satisfiability (k-SAT) or simply satisfiability (SAT), when each clause contains exactly k=3 and k=2 literals respectively. 2-SAT is P while 3-SAT is NP Complete.

Which of the following statements are TRUE?

The problem of determining whether there exists a cycle in an undirected graph is in P.

The problem of determining whether there exists a cycle in an undirected graph is in NP.

If a problem A is NP-Complete, there exists a non-deterministic polynomial time algorithm to solve A.

(a) 1, 2 and 3

(b) 1 and 2 only

(c) 2 and 3 only

(d) 1 and 3 only [GATE 2013]

Ans. (a) 1, 2 and 3

Explanation:

We can either use BFS or DFS to find whether there is a cycle in an undirected graph. For example, see DFS based implementation to detect cycle in an undirected graph. The time complexity is O (V+E) which is polynomial.

If a problem is in P, then it is definitely in NP (can be verified in polynomial time).

True.

Let SHAM3 be the problem of finding a Hamiltonian cycle in a graph G=(V,E) with V divisible by 3 and DHAM3 be the problem of determining if a Hamiltonian cycle exists in such graphs. Which one of the following is true?

(a) Both DHAM 3 and SHAM 3 are NP-hard

(b) SHAM 3 is NP-hard, but DHAM 3 is not

(c) DHAM 3 is NP-hard, but SHAM 3 is not

(d) Neither DHAM 3 nor SHAM 3 is NP-hard [GATE 2006]

Ans. (a) Both DHAM 3 and SHAM 3 are NP-hard

Explanation: The problem of finding whether there exist a Hamiltonian Cycle or not is NP Hard and NP Complete Both. Finding a Hamiltonian cycle in a graph G=(V, E) with V divisible by 3 is also NP Hard.

Consider the following two problems on undirected graphs

α: Given G(V, E), does G have an independent set of size | V | - 4?

β: Given G(V, E), does G have an independent set of size 5?

Which one of the following is TRUE?

(a) α is in P and β is NP-complete

(b) α is NP-complete and β is in P

(c) Both α and β are NP-complete

(d) Both α and β are in P [GATE 2005]

Ans. (c) Both α and β are NP-complete

Explanation: Graph independent set decision problem is NP Complete.

Consider the following two problems of graph.

Given a graph, find if the graph has a cycle that visits every vertex exactly once except the first visited vertex which must be visited again to complete the cycle.

Given a graph, find if the graph has a cycle that visits every edge exactly once. Which of the following is true about above two problems:

(a) Problem 1 belongs NP Complete set and 2 belongs to P

(b) Problem 1 belongs to P set and 2 belongs to NP Complete set

(c) Both problems belong to P set

(d) Both problems belong to NP complete set [GATE 2015]

Ans. (a) Problem 1 belongs NP Complete set and 2 belongs to P

Explanation: Problem 1 is Hamiltonian Cycle problem which is a famous NP Complete problem.

Problem 2 is Euler Circuit problem which is solvable in Polynomial time.

Consider two decision problems Q1, Q2 such that Q1 reduces in polynomial time to 3-SAT and 3-SAT reduces in polynomial time to Q2. Then which one of the following is consistent with the above statement?

(a) Q1 is in NP, Q2 is NP hard

(b) Q2 is in NP, Q1 is NP hard

(c) Both Q1 and Q2 are in NP

(d) Both Q1 and Q2 are in NP hard [GATE 2015]

Ans. (a)Q1 is in NP, Q2 is NP hard

Explanation:

Q1 reduces in polynomial time to 3-SAT

Q1 is in NP

3-SAT reduces in polynomial time to Q2

Q2 is NP Hard. If Q2 can be solved in P, then 3-SATcan be solved in P, but 3-SAT is NP-Complete, that makes Q2 NP Hard

Appendix

Previous Year Paper

B.E. 5th Semester (CSE) Examination,

December-2012

ANALYSIS AND DESIGN OF ALGORITHMS

Paper-CSE-305-E

Time allowed: 3 hours] [Maximum marks: 100

Note: Attempt any five questions. All questions carry equal marks.

(a) What do you mean by Asymptotic Notation? Explain various types of asymptotic notations.

(b) What do you mean by Quick Sort? Explain its time complexity.

(a) What do you mean by Strassen's Matrix Multiplication? Explain its complexity.

(b) Sort the following list of elements using Heap Sort.

20, 30, 10, 5, 7, 8, 17, 90, 80, 70, 65, 76, 88, 22, 55, 89

What do you mean by knapsack problem? How it is solved by Greedy approach? Explain with suitable example.

(a) What do you mean by Optimal Binary Search Tree? Explain with suitable example.

(b) What is travelling salesman problem? How it is solved by using dynamic programming?

(a) What is Graph coloring? Explain with its algorithm. Also give some suitable example.

(b) What is difference between backtracking and dynamic programming? How backtracking is used for solving the 8 Queen problem?

(a) Explain some principles for efficiency consideration.

(b) What do you mean by branch and bound strategy? How it solves the all pair shortest path problem?

(a) Explain the Cook's theorem in detail.

(b) Differentiate between NP hard and NP complete problems. Explain some NP Hard problems.

Write short notes on the following:

(a) Binary Search

(b) Merge sort

(c) Job sequencing with dead Line

(d) Least cost searching

B. Tech. 4th Sem.

End Term Examination - May-June, 2014

ANALYSIS & DESIGN OF ALGORITHMS

Paper: ETCS-204

Time: Three Hours] [Maximum Marks: 75

Note: Attempt five questions including Q. no. 1 which is compulsory. Select one question from each unit.

(a) Which is Quick sort? Show its functioning on data. (5×5=20)

(b) Explain the elements of Greedy strategy.

(c) Define Depth first search.

(d) Write Floyd-Warshall algorithm.

(e) What is string matching? Name some algorithms for string matching.

Unit-I

Write short note on the following: (12.5)

(a) Substitution method.

(b) Iteration method.

(a) What will be the complexity of insertion sort, quick sort and merge sort on following sequence.

Sequence 1 - 1, 2, 3, 4, 5, Sequence - 5, 4, 3, 2, 1

(b) Can we improve the worst case complexity of quick sort O (n ²) further? If yes then give its algorithm. (12.5)

Unit-II

(a) Write not eon the optimal binary search tree problems.

(b) What is matrix chain multiplication? Explain. (12.5)

What are Huffman codes? Why we use them? Discuss in detail. (12.5)

Unit-III

Discuss in details any one algorithm for finding the cost spanning tree. (12.5)

Explain Dijkstra's and Bellman Fort algorithm for finding single source shortest path in detail. (12.5)

What is NP-Completeness? Discuss any five NP-Complete problems in detail. (12.5)

B. Tech. (CSE) 6th Sem.

Examination - May, 2015

ANALYSIS & DESIGN OF ALGORITHMS

Paper: CSE-306-F

Time: Three Hours] [Maximum Marks: 100

Before answering the questions, candidates should ensure that they have been supplied the correct and complete question paper. No complaint in this regard, will be entertained after examination.

Note: Attempt five questions, selecting one question from each section and question number one is compulsory.

(a) Which function grows faster e n or 2 n ? Justify your answer. 2

(b) Analyze the various cases for Binary Search's complexity. 2

(c) Using big-O notation, state time and space complexity of quick-sort. 2

(d) What is the difference between greedy and dynamic approach? 2

(e) Can the master method be applied to solve recurrence: 2

T(n) =4T (n/2) + n ² log n ?

Why or why not?

(f) What are three properties of NP-Complete problem? 2

(g) Explain any Branch-and-Bound technique. 4

(h) Discuss Hamiltonian cycles with example. 4

SECTION - A

(a) Explain strassen's matrix multiplication with example. 10

(b) Write algorithms for Union & Find operations for disjoint sets. 10

(a) What is Merge-sort? Write a recursive algorithm for same and show that its running time is O(n log n). 12

(b) Design a Divide-&-Conquer algorithm for finding minimum and maximum element of 'n' numbers using no more than 3n/2 comparisons. 8

SECTION - B

(a) Use Dijkstra algorithm to find single source shortest path's for following graph taking vertex 'A' as the source 12

(b) Write Kruskal's algorithm for finding minimum spanning tree of an undirected graph. 8

(a) Explain Traveling Salesperson Problem. 8

(b) Set n=7; (p1, p2……., P7)=(3, 5, 20,18,1, 6, 30) and (d1, d2,…….... d7)=(1,3, 4, 3, 2, 1/2).

What is the solution generated by Job sequencing algorithm for given problem? 12

SECTION - C

(a) Discuss how 8-Oueen's Problem is solved through Backtracking. 10

(b) Explain LC Branch-&-Bound with example. 10

(a) Write Backtracking algorithm to find chromatic number of a given graph. 10

(b) What is O/l Knapsack problem? Solve this problem vising Branch-&-Bound method taking suitable example. 10

SECTION - D

(a) Giving suitable example prove that travelling Salesperson Problem is NP- hard. 10

(b) Show that clique decision problem is NP-hard. 10

Write short notes on:

(a) Difference between deterministic and non-deterministic algorithms. 6

(b) NP-hard and NP-complete problems. 6

(c) Cook's theorem. 8

B.E./B.Tech. 4th Sem.

Degree Examination - April/May, 2015 (Regulation 2013)

Computer Science and Engineering

Design and Analysis of Algorithms

(Common to Information Technology)

Paper: CS-6402

Time: Three Hours] [Maximum Marks: 100

Answer ALL questions

PART A

(10×2=20)

Write an algorithm to find the number of binary digits in the binary representation of a positive decimal integer.

Write down the properties of asymptotic notations.

Design a brute-force algorithm for computing the value of a polynomial p (x)= a n x n + a n -1 x n -1 + … + a 1 x … a 0 at a given point x 0 and determine its worst-case efficiency class.

Derive the complexity of Binary Search algorithm.

Write down the optimization technique used for Warshall's algorithm. State the ruels and assumptions which are implied behind that.

List out the memory functions used under Dyanmic Programming.

What do you mean by 'perfect matching' in bipartite graphs?

Define flow 'cut'.

How NP-Hard problems are different from NP-Complete?

Define hamiltonian Circuit problem.

PART B

(5×16=80)

(a) If you have to solve the searching problem for a list of n numbers, how can you take advantage of the fact that the list is known to be sorted? Give seaparate answers for

(i) lists represented as arrays.

(ii) lists represented as linked lists.

Compare the time complexities involved in the analysis of both the algorithms. (16)

OR

(b) (i) Derive the worst case analysis fo Merge Sort using suitable illustrations. (8)

(ii) Derive a losse bound on the following equation:

$f(x)=30x^8 - 22x^7 + 14x^5 - 2x^4 - 4x^2 + x - 15$. (8)

(b) (i) Solve the following using brute-Force algorithm: (10)

Find whether the given string follows the specified pattern and return 0 or 1 accordingly.

Examples:

(1) Pattern: "abba", input: "redblueredblue" should return 1

(2) Pattern: "aaaa", input: "asdasdasdasd" should return 1

(3) Pattern: "aabb", input: "xyzabcxzyabc" should return 0

(ii) Explain the convex hull problem and the solution involved behind it. (6)

OR

(b) A pair contains two numbers, and its second number is on the right side of the first one in an array. The difference of a pair is the minus result while subtracting the second number from the first one. Implement a function which gets the maximal difference of all pairs in an array (using Divide and Conquer method). (16)

*(a) (i) Given the mobile numeric keypad. You can only press button that are up, left, right or down to the first number pressed to obtain the subsequent numbers. You are not allowed to press bottom row corner buttons (i.e. * and #). Given a number N, how many key strokes will be involved to press the given number. What is the length of it? Which dynamic programming technique could be used to find solution for this? Explain*

this step with the help of a pseudo code and derive its time complexity. (12)

(ii) How do you construct a minimum spanning tree using Kruskal's algorithm? Explain. (4)

(b) (i) Let A =(l /119, m /96, c /247, g /283, h /72, f /77, k /92, j /19} be the letters and its frequency of distribution in a text file. Compute a suitable Huffman coding to compress the data effectively. (8)

(ii) Write an algorithm to construct the optimal binary search tree given the roots r (i , j), $0 \le i \le j < n$. also prove that this could be performed in time O (n). (8)

(a) (i) Maximize p =2 x + 3 y + z (8)

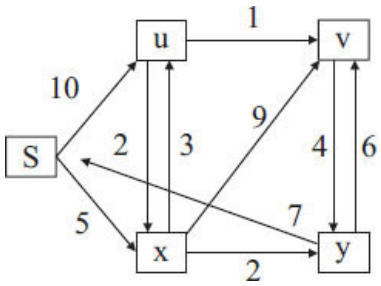
subject to $x + y + z \le 40$
$2x + y - z \ge 10$
$-y + z \ge 10$
$x > 0, y \ge 0, z \ge 0.$

(ii) Write down the optimality conditions and algorithmic implementation for fiding M-augmenting paths an bipartite graphs. (8)

OR

(b) (i) Briefly describe on the stable marriage problem. (6)

(ii) How do you compute maximum flow for the following graph using Ford-Fulkerson method? (10)



(a) (i) Suggest an approximation algorithm for traveling salesperson problem. Assume that the cost function satisfies the traignle inequality. (8)

(ii) Explain how job assignment problem could be solved, given n tasks and n agents where each agent has a cost to complete each task, using Branch and Bound technique. (8)

OR

(b) (i) The knight is placed on the first block of an empty board and moving according to the rules of chess, must visit each square exactly once. Solve the above problem using backtracting procedure. (10)

(ii) Implement an algorithm for Knapsack problem using NP-Hard approach. (6)

B.E./B.Tech. 3rd/4th Sem.

Degree Examination - May/June, 2016 (Regulation 2013)

Computer Science and Engineering

Design and Analysis of Algorithms

(Common to Information Technology)

Paper: CS-6402

Time: Three Hours] [Maximum Marks: 100

Answer ALL questions

PART A (10×2=20)

Give the Euclid's algorithm for computing ged (m, n).

Compare the orders of growth of n(n-1)/2 and n ² .

Give the general strategy of Divide and Conquer Method.

What is the closest - pair problem?

Define the Single Source Shortest Paths Problem.

State the assignment Problem.

What is a state space graph?

State Extreme Point Theorem.

Give the purpose of lower bound.

What is Euclidean minimum spanning tree problem?

PART B (5×16=80)

(a) (i) Give the definition and Graphical Representation of O-Notation.

(ii) Give the algorithm to check whether all the elements in a given array of a elements are distinct. Find Worst case complexity of the same.

OR

(b) Give the recursive Algorithm for finding the number of binary digits to n's binary representation, where n is a positive decimal integer. Find the recurrence relation and complexity.

(a) State and Explain the Merge Sort algorithm and Give the recurrence relation and efficiency.

OR

(b) Explain the method used for performing Multiplication of two large integers. Explain how Divide Conquer Method can be used to solve the same.

(a) Discuss about the algorithm and Pseudocode to find the Minimum Spanning Tree using Prim's Algorithm. Find the Minimum Spanning tree for the graph shown below:

And Discuss about the efficiency of the algorithm.

Find all the Solution to the travelling salesman problem (cities and distances shown below) by exhaustive search. Give the optimal solution. (16)

(i) Summarize the simplex method. (8)

(ii) State and prove Max-Flow Min-Cut Theorem. (8)

OR

Apply the shortest-augmenting-path algorithm to the network shown below. (16)

(a) Give any five undecidable problems and explain the famous halting problem. (16)

OR

(b) State the subset-sum problem and complete state-space tree of the backtracking algorithm applied to the instance A=(3, 5, 6, 7) and d=15 of the subset-sum problem. (16)

B.E./B.Tech. 3rd/4th Sem.

Degree Examination - Nov./Dec., 2015 (Regulation 2013)

Computer Science and Engineering

Design and Analysis of Algorithms

(Common to Information Technology)

Paper: CS-6402

Time: Three Hours] [Maximum Marks: 100

Answer ALL questions

PART A (10×2=20)

The (log n)th smallest number of a unsorted numbers can be determined O(n) average-case time (True/False).

Write the recursive Fibonacci algorithm and its recurrence relation.

Give the mathematical notation to determine if a convex direction is towards left or right and write the algorithm.

Prove that any comparison sort algorithm requires W (n log n) comparisons in the worst case.

State how Binomial Coefficient is computed?

What is the best algorithm suited to identify the topography for a graph. Mention its efficiency factors.

Determine the Dual linear program for the following LP.

Maximise $3a + 2b + c$
Subject to,

$$\begin{aligned} 2a + b + c &<= 3 \\ a + b + c &<= 4 \\ 3a + 3b + 6c &<= 6 \\ a, b, c &>= 0. \end{aligned}$$

Define Network Flow and Cut.

Draw the decision tree for comparison of three values.

Depict the proof which says that a problem 'A' is no harder or no easier than problem 'B'.

PART B (5×16=80)

(a) (i) Write the Insertion sort algorithm and estimate its running time. (8)

(ii) Find the closest asymptotic tight bound by solving the recurrence equation T(n)=8T (n/2) + n2 with (T(1)=1) using Recursion tree method. [Assume that T(1) e Θ (1)]. (8)

(b) (i) Suppose W satisfies the following recurrence equation and base case (where c is a constant): W(n)=c.n + W(n/s) and W(1)=1. What is the asymptotic order of W(n). (6)

(ii) Show how to implement a stack using two queues. Analyze the runing time of the stack operations. (10)

(a) (i) Write the algorithm to perform Binary Search and compute its run time complexity. (8)

(ii) Compute the multiplication of given two matrices using Strassen's matrix multiplication method: (8)

$$A = \begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

(b) (i) Write down the algorithm to construct a convex hull based on divide and conquer strategy. (8)

(ii) Find the optimal solution to the fractional knapsack problem with given data: (8)

Item	Weight	Benefit
A	2	60
B	3	75
C	4	90

(a) (i) The binary string below is the title of a song encoded using Huffman codes.

00110001011111011001110111011000001001110100101 01.

Given the letter frequencise listed in the table below, build the Huffman codes and use them to decode the title. In cases where there are multiple "greedy" choices, the codes are assembled by combining the first letters (or groups of letters) from left to right, in the order given in the table. Also, the codes are assigned by labeling the left and right branches of the prefix/code tree with '0' and '1', respectively. (10)

Letter	a	h	v	w	''	e	t	1	o
Frequency	1	1	1	1	2	2	2	3	3

(b) (i) Write and analyse the Prim's Algorithm. (8)

(ii) Consider the following weighted graph.

Give the list of edges in the MST in the order that Prim's algorithm inserts them. Start Prim's algorithm from vertex A.

(a) (i) Use Simplex to solve the farmers problem given below:

A farmer has a 320 acre farm on which he plants two crops: rice and wheat. For each acre of rice planted, his expenses are 50 and for each acre of wheat planted, his expenses are 100. Each acre of rice requires 100 quintals of storage and yields a profit of 60; each acre of wheat requires 40 quintals of storage and yields a profit of 90. If the total amount of storage space available is 19,200 quintals and the farmer has only '20,000 on

hand, how many acres of each crop should he plant in order to maximize his profit? What will his profit be if he follows this strategy? (12)

(ii) Write the procedure to Initialize Simplex which determines if a linear program is feasible or not? (4)

(b) (i) Illustrate the workings of the maximum matching algorithm on the following weighted tree. (12)

(ii) Explain Max-Flow Problem. (4)

(a) (i) Using an example prove that, satisfiability of boolean formula in 3-Conjunctive Normal Form is NP-complete. (12)

(ii) State the relationships among the complexity class algorithms with the help of neat diagrams. (4)

OR

(b) (i) Show that the hamiltonian Path problem reduces to the hamiltonian Circuit Problem and vice versa. (10)

(ii) What is an approximation algorithm? Give example. (6)

B.E./B.Tech. 5th Sem.

Degree Examination-

Design and Analysis of Algorithms

(Common to Information Technology)

Paper: A2099

Time: Three Hours] [Maximum Marks: 60

Instruction to Candidates

SECTION-A is COMPULSORY consisting of TEN questions carrying TWO marks each.

SECTION-B contains FIVE questions carrying FIVE marks each and students have to attempt any FOUR questions.

SECTION-C contains THREE questions carrying TEN marks each and students have to attempt any TWO questions.

SECTION A

Write briefly:

(a) What is asymptotic efficiency of algorithm?

(b) Define recurrence.

(c) State valid shift with reference to string matching.

(d) State an appication of FFT.

(e) Give example of NP complete problem.

(f) Define DFS for graphs.

(g) Define a heap.

(h) Define randomization.

(i) Give the complexity of heapsort.

(j) State approximation technique.

SECTION B

Compare the performance of insertion sort versus mergsort.

Write a short note on Convex hull.

Differentiate between NP hard and NP complete problems.

Write a short note on greedy strategy to solve a problem.

What is advantage of binary search over linear search? Also state limitations of binary search.

SECTION C

Write Strassen's algorithm for matrix multiplication.

Write Dijkstra's algorithm for single source shortest path problem on weighted directed graph.

Differentiate between sorting based on different design techniques.

B.E./B.Tech. 5th Sem.

Degree Examination -

Design and Analysis of Algorithms

(Common to Information Technology)

Paper: A2099

Time: Three Hours] [Maximum Marks: 60

Instruction to Candidates

SECTION-A is COMPULSORY consisting of TEN questions carrying TWO marks each.

SECTION-B contains FIVE questions carrying FIVE marks each and students have to attempt any FOUR questions.

SECTION-C contains THREE questions carrying TEN marks each and students have to attempt any TWO questions.

SECTION A

Write briefly:

(a) What is asymptotic efficiency of algorithm?

(b) Define recurrence.

(c) State valid shift with reference to string matching.

(d) State an appication of FFT.

(e) Give example of NP complete problem.

(f) Define DFS for graphs.

(g) Define a heap.

(h) Define randomization.

(i) Give the complexity of heapsort.

(j) State approximation technique.

SECTION B

Compare the performance of insertion sort versus mergsort.

Write a short note on Convex hull.

Differentiate between NP hard and NP complete problems.

Write a short note on greedy strategy to solve a problem.

What is advantage of binary search over linear search? Also state limitations of binary search.

SECTION C

Write Strassen's algorithm for matrix multiplication.

Write Dijkstra's algorithm for single source shortest path problem on weighted directed graph.

Differentiate between sorting based on different design techniques.

B.Tech.

(SEM. V) (ODD SEM.) Theory

Examination, 2014-15

Design and Analysis of Algorithms

Time: Three Hours] [Maximum Marks: 60

Instruction to Candidates

SECTION-A is COMPULSORY consisting of TEN questions carrying TWO marks each.

SECTION-B contains FIVE questions carrying FIVE marks each and students have to attempt any FOUR questions.

SECTION-C contains THREE questions carrying TEN marks each and students have to attempt any TWO questions.

SECTION A

Attempt any four parts of the following: 5 × 4=20

(a) Solve the following recurrences:

(i) T(n)=T(n/2) + T(n/4) + T(n/8) + n

(ii) T(n)=T(÷n) + O(lg n)

(b) What is the time complexity of counting sort?

Illustrate the operation of counting sort on array

A={1,6,3,3,4,5,6,3,4,5}

(c) Describe the properties of red Black tree. Show that Red Black Tree with n internal nodes has height at most 2lg(n+1).

(d) Discuss the complexity of Max-Heapify and Build-Max Heap procedures.

(e) Discuss asymptotic notations in brig.

(f) Discuss the best case and worst case complexities of quick sort algorithm in detail.

SECTION B

Attempt any two parts of the following: 10 × 2=2

(a) What are the advantages of Red Black Tree over Binary Search Tree? Write algorithms to insert a key in a red black tree. insert the following sequence of information in an empty red black tree 1, 2, 3, 4, 5, 5.

(b) Define the binomial heap in detail. Write an algorithm for performing the union operation of two binomial heaps and also explain with suitable example.

(c) How B-Tree differs with other tree structures. Insert the following information F, S, Q, K, C, L, V, W, M, R, P, A, D, Z, E into an empty B-Tree with degree t=2.

SECTION C

Attempt any two parts of the following: 10 × 2=20

(a) What do you mean by minimum spanning tree? Write an algorithm for minimum spanning tree that may generate multiple forest tree and also explain with suitable example.

(b) Describe in detail the Stresses's Matrix Multiplication algorithms based on divide and conquer strategies with suitable example.

(c) Given a weighted graph G=(V, E) with source s and weight function W: E Æ R, then write an algorithm to solve a single source shortest path problem whose complexity is O (VE). Apply the same on the following graph.

Attempt any two parts of the following: 10 × 2=20

(a) Differentiate between Dynamic programming and Greedy approach. What is 0/1 knapsack problem? Solve the following instance using Dynamic programming, write the algorithm also. Knapsack Capacity=10 P=< 1,6,18,22,28,> and w=<1,2,5,6,7>.

(b) Differentiate between Backtracking and Branch and Bound approach. Write an algorithms for sum subset problem using back tracking approach. Find all possible solution for following instance using same if m=30, S=< 1,2,5,7,8,10,15,20,25 >.

(c) Define TSP problem in detail. Find the solution for the following instance of TSP problem using branch and bound.

Attempt any two parts of the following: 10 × 2=20

(a) Define different complexity classes in detail with suitable example. Show that TSP problem is NP Complete.

(b) Describe approximation Algorithm in detail. What is the approximation ratio? Show that vertex cover problem is 2 approximate.

(c) What is string matching algorithm? Write Knuth-Morris-Pratt algorithm and also calculate the prefix function for the pattern P=abababaca.

B.Tech.

(SEMESTER-V) Theory

Examination, 2012-13

Design and Analysis of Algorithms

Time: Three Hours] [Maximum Marks: 100

Instruction to Candidates

SECTION-A is COMPULSORY consisting of TEN questions carrying TWO marks each.

SECTION-B contains FIVE questions carrying FIVE marks each and students have to attempt any FOUR questions.

SECTION-C contains THREE questions carrying TEN marks each and students have to attempt any TWO questions.

Note: Answer all the Sections.

SECTION A

Attempt all question parts. 10 × 2=20

(A) Which of the following order of growth is correct?

(a) n ²

(b) nlog 2 n < n ³ < n!

(c) n < log 2 n < n ²

(d) n < 2n < n ³

(B) The order of time for creating a heap of size n is

(a) O(n)

(b) O(log n)

(c) n < log 2 n < n ²

(d) n < 2 n < n ³

(C) Quick sort exhibits its worst case behavior when the input data is in_.

(a) already sorted

(b) reverse sorted

(c) random

(d) do not have worst case

(D) Every internal node in a B-tree of minimum degree 2 can have _.

(a) 2, 3 or 4

(b) 1, 2 or 3

(c) 2, 4 or 6

(d) 0, 2 or 4

(E) The second largest number from a set of n distinct number can be found in

(a) O(n)

(b) O(1)

(c) O(n ²)

(d) O(log n)

(F) Back-Tracking and Branch-and-bound based solutions use_.

(a) Spanning Tree

(b) Decision Tree

(c) Binary Tree

(d) State-space Tree

(G) A function t(n) is to be in O(g(n)) if t(n)

(a) is boundary both above and below by some constant multiples of g(n)

(b) is bounded above by some constant multiple of g(n) for all n

(c) is bounded below by some constant multiple of g(n) for all large n

(d) is bounded above by some function of g(n)

(H) Consider the following graph. Which of the following is NOT the sequence of edges to the minimum spanning tree using Kruskal's algorithm?

(a) (b, e), (e, f), (a, c), (b, c), (f, g), (c, d)

(b) (b, e), (e, f), (a, c), (f, g), (b, c), (c, d)

(c) (b, e), (a, c), (e, f), (b, c), (f, g), (c, d)

(d) (b, e), (e, f), (b, c), (a, c), (f, g), (c, d)

(I) A Hamiltonian circuit is

(a) the shortest cycle through all vertices of a graph.

(b) the fastest cycle through distict vertices of a graph.

(c) a cycle passes through all the vertices of a graph exactly once excepts the start node.

(d) cycle through points which from the smallest polygon that contains all points of a set of points.

(J) NP is class of all decision problems whose randomly guessed solution can be verified in

(a) Derterministic polynomial time

(b) Nondetereministic polynomial time

(c) NP hard time

(d) NP complete time

SECTION B

Attempt any two parts of the following: 10 × 3=30

(a) (i) Describe the difference between average-case and worst-case analysis of algorithms, and give an example of an algorithm whose average-case running time is different from its worst-case time. (5)

(ii) How will you represent a max-heap sequentially? Explain with an example in the below given heap. (5)

(b) (i) Consider the following valid re-black tree, where R indicates a red node, B indicates a black node. Note that the black dummy sentinel leaf nodes are not shown. Show the resulting red-black tree after inserting key 3 into and deleting 15 from the original tree. (5)

(ii) Show any two legal B-Trees of minimum degree 3 that represent {1, 2, 3, 4, 5, 6}.

(c) (i) Suppose that undirected graph G=(V; E) has non-negative edge weights and these are raised by 1. Can the minimum spanning tree change? Can shortest paths? Justify proper example. (5)

(ii) Show all the steps of Strassen's matrix multiplication algorithm to multiply the following matrices. (5)

$$X = \begin{bmatrix} 3 & 2 \\ 4 & 8 \end{bmatrix} \text{ and } Y = \begin{bmatrix} 1 & 5 \\ 9 & 6 \end{bmatrix}$$

(d) (i) Consider the sum-of-subset problem, n=4, Sum=13, and wt 1 =3, wt 2 =4, wt 3 =5 and wt 4 =6. Find a solution to the problem using backtraking. Show the state-space tree leading to the solution. Also number the nodes in the order recursion calls. (6)

(ii) State the implicit and constraints of n-queens problem. (4)

(e) In the graph given below: (10)

(i) Write the triangle inequality algorithm to find solution for the Travelling Salesman problem.

(ii) Is the solution obtained from the algorithm optimal in all cases?

(iii) For the graph given above, apply, apply the algorithm starting from city A and obtain the solution. Properly indicate all the intermediate steps of execution of the algorithm.

SECTION C

Attempt all questions: 10 × 5=50

Attempt any two parts: (5 × 2=10)

(a) Solve the following recurrences using the Master method:

$T(1)=0$

$T(n)=9\ T(n/3) + n^3 \log n;\ n > 1$

(b) What is the minimum number of keys in a B-tree of order 32 and height 5?

(c) Given the items in the table below and a knapsack with weight limit 100, what is the solution to this problem?

Item ID	Weight	Value	Value/Weight
A	100	40	0.4
B	50	35	0.7
C	40	20	0.5
D	20	4	0.2
E	10	10	1
F	10	6	0.6

Attempt any one part: 10 × 1=10

(a) Write the merge sort algorithm for sorting a set of n points. Draw the recursion tree for n=13.

(i) How many levels are there in the tree?

(ii) How many comparisons are done at each level in the worst case?

(iii) What is the total number if comparisons needed?

(iv) Generalize (i) to (iii) for any n (assume n is power of 2) in terms of O().

(b) Write the algorithm for deleting an element from a binomial-heap. Show the binomial-heap that results when the element 21 is removed from H given below:

Attempt any one part: 10 × 1=10

(a) Suppose Dijkstra's algorithm is run on the following graph, starting at node A,

(i) Draw a table showing the intermediate distance values of all the nodes at each iteration of the algorithm.

(ii) Show the final shortest path tree.

(b) (i) What is the purpose of Floyd-Warshall's algorithm?

(ii) Write the pseudo-code of the algorithm.

(iii) What is the time complexity of the algorithm.

(iv) Suppose Floyed-Warshall's algorithm is run on the weighted, directed graph shown below, show the value of the matrices that result from each iteration in the algorithm:

Attempt any one part: 10 × 1=10

(a) Define vertex cover. What is vertex cover problem? Provide the approximation algorithm for vertex cover problem. Run the algorithm on the graph given below and obtain the solution

(b) Let P=rrllrrll be pattern and T=irrrlrrlllrrrrlrrlrrllrlrrlrrlr be a tex in a string matching problem:

(i) How many shifts (both valid and invailid) will be made by the Naive String matching algorithm?

(ii) Provide the algorithm to compute the transition function for a string matching automation.

(iii) Find out the state transition diagram for the automation to accept the pattern P given above.

Attempt any one parts: 5 × 2=10

(a) You are given the following iterative algorithm:

```
1.  Mystery(A[0, n–1])
2.  //Input: An array A[0, n–1] of n real numbers
3.  for (i = 0; i <= n – 2; i ++) {
4.  for (j = i + 1; j <= n – 1; j++) {
5.  if (A[i] = = A[j])
6.  return false;
7.      }
8.  }
9.  return true;
```

(i) What does this algorithm compute?

(ii) What is the best-case time complexity of the algorithm, as a function of n?

(iii) What is the worst-case time complexity of the algorithm, as a function of n?

(b) What is Bellman-ford algorithm? Provide pseudo-code of the algorithm and derive its time complexity.

(c) Prove that circuit satisfiability problem belongs to the class NP.

B.Tech.

(SEM. V) ODD Semester

Theory Examination, 2012-13

Design and Analysis of Algorithms

Time: Three Hours] [Maximum Marks: 100

Instruction to Candidates

SECTION-A is COMPULSORY consisting of TEN questions carrying TWO marks each.

SECTION-B contains FIVE questions carrying FIVE marks each and students have to attempt any FOUR questions.

SECTION-C contains THREE questions carrying TEN marks each and students have to attempt any TWO questions.

Note: Answer all the Sections.

SECTION A

Attempt any four parts of the following:

*(a) If f(n)=100 * 2 n + n ⁵ n, then show that f(n)=O(2 n).*

(b) Consider the following function:

```
int SequentialSearch(int A[], int & x, int n)
{
int i;
for (int i=0; i < n && a[i] !=x; i++)
if (i==n) return-1;
rerturn i;
}
```

Determine the average and worst case time complexity of the function sequentialSearch.

(c) Consider a polynomial:

*P(x)=(….. (c n * x + c n-1)* x + c n-2)x + c n-3)x….)* x + c 0 .*

Estimate the time complexity to evaluate the polynomial

P(x).

*(d) Show that 10 * log 2 n + 10 ⁵ =q(log 2 n).*

(e) What do you mean by recursion? Explain your answer with an example.

(f) Determine the best case time complexity of merge sort algorithm.

SECTION B

Attempt any two parts of the following:

(a) Define a red-black tree. Explain the insertion operation in a red-black tree.

(b) Define a B-tree of order m. Explain the searching operation in a B-tree.

(c) What is a Fibonacci heap? Discuss the applications of Fibonacci heaps.

SECTION C

Attempt any two parts of the following:

(a) Solve the recurrence

T(n)=T(n - 1) + T(n - 2) + 1, when T(0)=0 and T(1)=1.

(b) Explain Kruskal's algorithm to find minimum cost spanning tree of an n-vertex undirected network.

(c) What is 0/1-knapsack problem? Does greedy method effective to solve the 0/1-knapsack problem?

Attempt any two parts of the following:

(a) What is dynamic programming? Explain your answer with an example.

(b) Describe traveling salesman problem (TSP). Show that a TSP can be solved using backtracking method in the exponential time.

(c) Discuss n-Queen problem.

Write short notes on any two of the for following:

(a) Randomized Algorithem.

(b) NP-complete problems.

(c) String matching algorithms.

B.Tech.

(SEM. V) ODD Semester

Theory Examination, 2013-14

Design and Analysis of Algorithms

Time: Three Hours] [Maximum Marks: 100

Instruction to Candidates

SECTION-A is COMPULSORY consisting of TEN questions carrying TWO marks each.

SECTION-B contains FIVE questions carrying FIVE marks each and students have to attempt any FOUR questions.

SECTION-C contains THREE questions carrying TEN marks each and students have to attempt any TWO questions.

Note: - (1) All questions are compulsory.

(2) Each questions carriers equal marks.

SECTION A

Attempt any four parts of the following: (5 × 4=20)

(a) Consider the recurrences

T (n)=3 T (n/3) + cn, and

T (n)=5 T (n/4) + n ² where c is constant and n is the number of inputs of inputs. Find the asymptotic bounds.

(b) What do you mean by algorithm? Write the characteristics of algorithm.

(c) Sort the following array using heap-sort techniques:

{5, 13, 2, 25, 7 17, 20, 8, 4}. Discuss its case and average case time complexities.

(d) Describe any one of the following sorting techniques:

(i) Selection sort

(ii) Insertion sort.

(e) What do you understand by asymptotic notations?

Describe important types of asymptotic notations.

(f) What is recursion tree? Describe.

SECTION B

Attempt any two parts of the following: (10 × 2=20)

(a) Explain red-black tree. Show steps of inserting the keys 41, 38 31 12, 19, 8 into initially empty red-black tree.

(b) Write the characteristics of a B-Tree of order m. Create B-Tree of order 5 from the following lists of data items:

20, 30, 35, 85, 10, 55, 60, 25, 5, 65, 70, 75, 15, 40, 50, 80, 45.

(c) What is a binomial heap? Describe the union of binomial heap.

SECTION C

Attempt any two parts of the following: (10 × 2=20)

(a) Describe and compare following algorithms to determine the minimum cost spanning tree:

(i) Kruskal's algorithm

(ii) Prime's algorithm.

(b) What is an optimization problem? How greedy method can be used to solve the optimization problem?

(c) What is matrix chain multiplication problem? Describe a solution for matrix chain multiplication problem.

Attempt any parts of the following: (10 × 2=20)

(a) Write an algorithm to find shortest path between all pairs of nodes in a given graph.

(b) Write short notes on the following:

(i) n-Queen problem

(ii) Graph coloring.

(c) What is Travelling Salesman Problem (TSP)? Discuss at least one approach used to solve the problem.

Attempt any two parts of the following: (10 × 2=20)

(a) Discuss the problem classes P, NP and NP-complete.

(b) What is FFT (fast Fourier Transformation)? How the reclusive FFT procedure works? Explain.

(c) Write short notes on Randomized algorithms.

B.Tech. (CSE) 6th Semester Examination,

May-2015

ANALYSIS AND DESIGN OF ALGORITHMS (24362)

PAPER-CSE-306-F

Time: 3 hours] [Maximum marks: 100

Before answering ht equestions, candidates should ensure that they have been supplied the correct and complete question paper. No complaint in this regard, will be entertained after examination.

Note: Attempt five questions, selecting one question from each section and question number one is compulsory.

(a) Which function grows faster e n or 2 n ? Justify your answer. 2

(b) Analyze the various cases for Binary Search's complexity. 2

(c) Using big-O notation, state time and space complexity of quick-short. 2

(d) What is the difference between greedy and dynamic approach? 2

(e) Can the master method be applied to solve recurrence:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2 \log n \text{ ? Why or why not ?}$$

(f) What are three properties of NP-Complete problem? 2

(g) Explain any Branch-and-Bound technique. 4

(h) Discuss Hamiltonian cycles with example. 4

SECTION - A

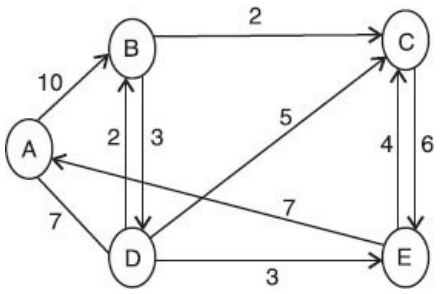
(a) Explain strassen's matrix multiplication with example. 10

(b) Write algorithms for Union & Find operations for disjoint sets. 10

(a) What is Merge-sort? Write a recursive algorithm for same and that its running time is O(n log n). 12

(b) Design a Divided-&-Conquer algorithm for finding minimum and maximum element of 'n' numbers using no more than 3n/2 comparisons. 8

(a) Use Dijkstra algorithm to find single sources shortest path's for following graph talking vertex 'A' as the source 12



(b) Write Kruskal's algorithm for finding minimum spanning tree of an undirected graph. 8

(a) Explain Traveling Salesperson Problem. 8

(b) Set n=7; (p1, p2……., p7)=(3, 5, 20, 18, 1, 6, 30) and (d1, d2,…….. d7)=(1, 3, 4, 3, 2, 1, 2). What is the solution generated by job sequencing algorithm for given problem? 12

SECTION - C

(a) Discuss how 8-Queen's Problem is solved through Backtracking. 10

(b) Explain LC Branch-& Bound with example. 10

(a) Write Backtracking algorithm to find chromatic number of a given graph. 10

(b) What is 0/1 Knapsack problem? Solve this problem using Branch-& Bound method tacking suitable example. 10

SECTION - D

(a) Giving suitable example prove that travelling Salesperson Problem is NP- hard. 10

(b) Show that clique decision problem is NP-hard. 10

Write short notes on:

(a) Difference between deterministic and nondeterministic algorithms. 6

(b) NP-hard and NP-complete problems. 6

(c) Cook's theorem. 8

B.Tech.

(SEM. V) Theory

Examination, 2015-16

Design and Analysis of Algorithms

Time: Three Hours] [Maximum Marks: 100

Instruction to Candidates

SECTION-A is COMPULSORY consisting of TEN questions carrying TWO marks each.

SECTION-B contains FIVE questions carrying FIVE marks each and students have to attempt any FOUR questions.

SECTION-C contains THREE questions carrying TEN marks each and students have to attempt any TWO questions.

SECTION A

Note: All questions are compulsory

Attempt all parts. All parts carry equal marks. Write answer of all part in short: (2 × 10=20)

(a) Justify why Quick sort is better than Merge sort?

(b) What is priority queue?

(c) Find out Hamiltonian cycles in complete graph having 'n' vertices.

(d) Explain binomial heap with properties.

(e) Explain element searching techniques using divide and conquer approach.

(f) Find the subsets-of following problem. Given total elements are (S)={4,2,7,6,8} and maximum SUM is (X)=8.

(g) Explain dynamic programming. How it is different from greedy approach?

(h) Solve the given recurrence T(n)=4T(n/4) + 4

(i) Differences back tracking and branch & bound programming approach.

(j) Explain the P, NP and NP-complete in decision problems.

SECTION B

Note: Attempt any five questions from this section. (10 × 5=50)

Explain insertion in Red Black Tree. Show steps for inserting 1, 2, 3, 4, 5, 6, 7, 8 & 9 into empty RB tree.

Discuss knapsack problem with respect to dynamic programming approach. Find the optimal solution for given problem, w (weight set)={5, 10, 15, 20} and W (Knapsack size)=8.

What is heap sort? Apply heap sort algorithm for sorting 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Also deduce time complexity of heap sort.

Explain B-Tree and insert elements B, Q, L, F into B-Tree [Fig: 1] then apply deletion of elements F, M, G, D, B on resulting B-Tree.

Fig. 1

Write an algorithm for solving n-queen problem. Show the solution of 4 queen problem using backtracking approach.

Explain a greedy single source shortest path algorithm with example.

What is string matching algorithm? Explain Rabin-Karp method with examples.

Explain Approximation algorithms with suitable examples.

SECTION C

Note: Attempt any two questions from this section. (15 × 2=30)

What is Fibonacci heap? Explain CONSOLIDATE operation with suitable example fro Fibonacci heap.

What is minimum spanning tree? Explain Prim's Algorithm and find MST of graph [Fig: 2]

Fig. 2

Explain TSP (traveling sales person) problem with example. Write an approach to solve TSP problem.