VTU eNotes On

Analysis and Design of Algorithms

(Computer Science)

Chapter 1. Introduction

1.1 **Need for studying algorithms:** The study of algorithms is the cornerstone of computer science.It can be recognized as the core of computer science. Computer programs would not exist without algorithms. With computers becoming an essential part of our professional & personal life's, studying algorithms becomes a necessity, more so for computer science engineers.

Another reason for studying algorithms is that if we know a standard set of important

algorithms ,They further our analytical skills & help us in developing new algorithms for required applications

1.2 ALGORITHM

An algorithm is finite set of instructions that is followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. Input. Zero or more quantities are externally supplied.
2. Output. At least one quantity is produced.
3. Definiteness. Each instruction is clear and produced.
4. Finiteness. If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. Effectiveness. Every instruction must be very basic so that it can be carried out, in principal, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.

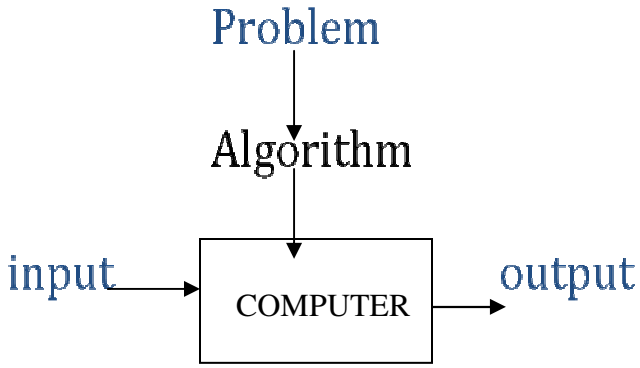

Fig 1.a.

An algorithm is composed of a finite set of steps, each of which may require one or more operations. The possibility of a computer carrying out these operations necessitates that certain constraints be placed on the type of operations an algorithm can include. The fourth criterion for algorithms we assume in this book is that they terminate after a finite number of operations.

Criterion 5 requires that each operation be effective; each step must be such that it can, at least in principal, be done by a person using pencil and paper in a finite amount of time. Performing arithmetic on integers is an example of effective operation, but arithmetic with real numbers is not, since some values may be expressible only by infinitely long decimal expansion. Adding two such numbers would violet the effectiveness property.

- Algorithms that are definite and effective are also called computational procedures.
- The same algorithm can be represented in same algorithm can be represented in several ways
- Several algorithms to solve the same problem
- Different ideas different speed

Example:

Problem:GCD of Two numbers m,n

Input specifiastion :Two inputs,nonnegative,not both zero

Euclids algorithm

-gcd(m,n)=gcd(n,m mod n)

Untill m mod n =0,since gcd(m,0) =m

Another way of representation of the same algorithm

Euclids algorithm

Step1:if n=0 return val of m & stop else proceed step 2

Step 2:Divide m by n & assign the value of remainder to r

Step 3:Assign the value of n to m,r to n,Go to step1.

Another algorithm to solve the same problem

Euclids algorithm

Step1:Assign the value of min(m,n) to t

Step 2:Divide m by t.if remainder is 0,go to step3 else goto step4

Step 3: Divide n by t.if the remainder is 0,return the value of t as the answer and stop,otherwise proceed to step4

Step4 :Decrease the value of t by 1. go to step 2

1.3 Fundamentals of Algorithmic problem solving

- Understanding the problem
- Ascertain the capabilities of the computational device
- Exact /approximate soln.
- Decide on the appropriate data structure
- Algorithm design techniques
- Methods of specifying an algorithm
- Proving an algorithms correctness
- Analysing an algorithm

Understanding the problem:The problem given should be understood completely.Check if it is similar to some standard problems & if a Known algorithm exists.otherwise a new algorithm has to be devised.Creating an algorithm is an art which may never be fully automated. An important step in the design is to specify an instance of the problem.

Ascertain the capabilities of the computational device: Once a problem is understood we need to Know the capabilities of the computing device this can be done by Knowing the type of the architecture,speed & memory availability.

Exact /approximate soln.: Once algorithm is devised, it is necessary to show that it computes answer for all the possible legal inputs. The solution is stated in two forms,Exact solution or approximate solution.examples of problems where an exact solution cannot be obtained are i)Finding a squareroot of number.

ii)Solutions of non linear equations.

Decide on the appropriate data structure:Some algorithms do not demand any ingenuity in representing their inputs.Someothers are in fact are predicted on ingenious data structures.A data *type* is a well-defined collection of data with a well-defined set of operations on it.A data *structure* is an actual implementation of a particular abstract data type. The Elementary Data Structures are

ArraysThese let you access lots of data fast. (good) .You can have arrays of *any* other da ta type. (good) .However, you cannot make arrays bigger if your program decides it needs more space. (bad) .

RecordsThese let you organize non-homogeneous data into logical packages to keep everything together. (good) .These packages do not include operations, just data fields (bad, which is why we need objects) .Records do not help you process distinct items in loops (bad, which is why arrays of records are used)

SetsThese let you represent subsets of a set with such operations as intersection, union, and equivalence. (good) .Built-in sets are limited to a certain small size. (bad, but we can build our own *set data type* out of arrays to solve this problem if necessary)

Algorithm design techniques: Creating an algorithm is an art which may never be fully automated. By mastering these design strategies, it will become easier for you to devise new and useful algorithms. Dynamic programming is one such technique. Some of the techniques are especially useful in fields other then computer science such as operation research and electrical engineering. Some important design techniques are linear, non linear and integer programming

Methods of specifying an algorithm: There are mainly two options for specifying an algorithm: use of natural language or pseudocode & Flowcharts.

A Pseudo code is a mixture of natural language & programming language like constructs. A flowchart is a method of expressing an algorithm by a collection of connected geometric shapes.

Proving an algorithms correctness: Once algorithm is devised, it is necessary to show that it computes answer for all the possible legal inputs .We refer to this process as algorithm validation. The process of validation is to assure us that this algorithm will work correctly independent of issues concerning programming language it will be written in. A proof of correctness requires that the solution be stated in two forms. One form is usually as a program which is annotated by a set of assertions about the input and output variables of a program. These assertions are often expressed in the predicate calculus. The second form is called a specification, and this may also be expressed in the predicate calculus. A proof consists of showing that these two forms are equivalent in that for every given legal input, they describe same output. A complete proof of program correctness requires that each statement of programming language be precisely defined and all basic operations be proved correct. All these details may cause proof to be very much longer than the program.

Analyzing algorithms: As an algorithm is executed, it uses the computers central processing unit to perform operation and its memory (both immediate and auxiliary) to hold the program and data. Analysis of algorithms and performance analysis refers to the task of determining how much computing time and storage an algorithm requires. This is a challenging area in which some times require grate mathematical skill. An important result of this study is that it allows you to make quantitative judgments about the value of one algorithm over another.

Another result is that it allows you to predict whether the software will meet any efficiency constraint that exits.

Performance analysis

There are any criteria upon which we can judge an algorithm for instance:

1. Does it do what we want to do?

2. Does it work correctly according to the original specifications to the task?

3. is there documentation that describes how to use it and how it works?

4. Are procedures created in such a way that they perform logical sub functions?

5. is the code readable?

The space complexity of an algorithm is the amount of memory it needs to run to completion. The time complexity of an algorithm is the amount of computer time it needs to run to completion.

Performance evaluation can be loosely divided into two major phases:

(1) A priory estimate and (2) a posteriori testing. We refer to these performance analysis and performance measurements respectively.

Space complexity

(1)A fixed part that is independent of characteristics (e.g., number, size) of the inputs and outputs this part typically includes the instruction space (i.e. space for code), space for simple variables and fixed size component variables (also called aggregate),space for constants and so on.

(2) A variable part that consist of space needed by component variable whose size is dependent on particular problem instance being solved, the space needed by referenced variables (to the extent that it depends on instance characteristics), and the recursion stack space (insofar and this space depends on the instance characteristics). The space requirement S(P) of an algorithm P may therefore be written as $S(P) = c + S_P$ (instance characteristics),where c is constant. When analyzing the space complexity of an algorithm, we concentrate solely on estimating S_P (instance characteristics). For any given problem, we

need first to determine which instance characteristics to use to measure the space requirement. Generally speaking our choices are related to the number and magnitude of the inputs to and outputs from the algorithm at times, more complexity measures of the interrelationship among the data times are used.

Time complexity

The time T (P) taken by a program P is sum of compile time and run time. The compile time does not depend on the instance characteristics. Also, we may assume that a compiled program will be run several time of a program. This run time is denoted by tp.

Because of many of the factor tp depends on are not known at the time of a program is conceived, it is reasonable to attempt only to estimate tp. If we knew the characteristics of a compiler to be used, we could

Proceed to determine the number of additions, subtractions, multiplication, divisions, compares, loads, stores and so on, that would be made by the code for P. So, we could obtain an expression for t_p (n) of the form

$t_p(n)=c_aADD(n)+c_sSUB(n)+c_mMUL(n)+c_dDIV(n)+\ldots.$

Where n denotes the instance characteristics ,and ca, c_s , c_m,c_d, and so on, respectively, denote the time needed for an addition ,subtraction, multiplication, division and so on, and ADD, SUB,MUL,DIV, and so on are the functions whose values are numbers of additions ,subtractions, multiplication, division and soon ,that are performed when code for P is used on an instance with characteristics n.

Obtaining such an exact formula is in itself an impossible task, since the time needed for addition, subtraction, multiplication, and so on, depend on the number being added, and subtract, multiplication and so on. The value of tp (n) for any given n can be obtained only experimentally. The program is typed, compiled, and run on a particular machine. The execution time is physically clocked, and tp (n) obtained. Even with this experimental approach, one could face difficulties. In a multiuser system, the execution time depends on such factors as system load, the number of other programs running on the computer at the time program P is run, the characteristics of these programs, and so on.

Given the minimal utility of determining the exact number of additions, subtraction, and so on, that are needed to solve a problem instance with characteristics given by n, we might as well lump all the operations together and obtain a cont for the total number of operations .We can go one more step further and count only the number of program steps.

A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics. For example, the entire statement

Return a+b+b*c+(a+b-c)/(a+b)+4.0;

Of Algorithm 1.5 could be regarded as a step since its execution time is independent of the instance characteristics (this statement is not strictly true , since the time for a multiply and divide generally depends on the numbers involved in the operation).

The number of steps any program statement is assigned depends on the kind of statement. For example comments count as zero steps; an assignment statement which does not involve calls any to other algorithms is encountered as one step; in an iterative statement such as **for**, **while** and **repeat until** statement, we consider the step count only for control part of the statement. The control parts for and while statements have the following forms:

for i=<expr>to<expr1> **do**

while (<expr>) **do**

each execution of the control part of a while statement is a step count equal to the number of step counts assignable to <expr> .the step count for each execution of control part of a **for** statement is one, unless the count attribute to <expr> and <expr1> are functions of the instance characteristics . In this latter case the first execution of the control part of the **for** has step count equal to the sum of counts for <expr> and <expr1>.remaining executions of the **for** statement have a step count of one; and so on.

We can determine number of steps needed by program to solve a particular problem instance in one of the two ways. in the first method we introduce a new variable ,count ,into the program .this is the global variable with initial value 0.statement to increment count by appropriate amount are introduced into the program. this is done so that each time a statement in the original program is executed ,count is incremented by step count of that statement.

1.4 Important Problem Types

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
2. The output is a permutation, or reordering, of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956.[1] Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2004). Sorting problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds.

Searching : In computer science, a **search algorithm**, broadly speaking, is an algorithm for finding an item with specified properties among a collection of items. The items may be stored individually as records in a database; or may be elements of a search space defined by a mathematical formula or procedure, such as the roots of an equation with integer variables; or a combination of the two, such as the Hamiltonian circuits of a graph.**Searching** algorithms are closely related to the concept of dictionaries. Dictionaries are data structures that support search, insert, and delete operations. One of the most effective representations is a hash table. Typically, a simple function is applied to the key to determine its place in the dictionary. Another efficient search algorithms on sorted tables is binary search

VTU eNotes On Analysis and Design of Algorithms (Computer Science)