JEPPIAAR INSTITUTE OF TECHNOLOGY

"Self-Belief | Self Discipline | Self Respect"

DEPARTMENT

OF

COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

CS8451 – DESIGN AND ANALYSIS OF ALGORITHM

(Regulation 2017)

Year/Semester: II/04 CSE

2020 – 2021

Prepared by

Ms.S.Sudha Mercy

Assistant Professor/CSE

UNIT I INTRODUCTION

Notion of an Algorithm – Fundamentals of Algorithmic Problem Solving – Important Problem Types – Fundamentals of the Analysis of Algorithm Efficiency – Analysis Framework – Asymptotic Notations and its properties – Mathematical analysis for Recursive and Non-recursive algorithms.

1.1 Algorithm

An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

These examples will help us to illustrate several important points:

- The nonambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.

1.2 FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

These solutions are not answers but specific instructions for getting answers. It is this emphasis on precisely defined constructive procedures that makes computer science distinct from other disciplines. In particular, this distinguishes it from theoretical mathematics, whose practitioners are typically satisfied with just proving the existence of a solution to a problem and, possibly, investigating the solution's Properties

Understanding the Problem

- From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given.
- Read the problem's description carefully and ask questions, do a few small examples by hand, think about special cases.

Ascertaining the Capabilities of the Computational Device

- Algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine—a computer architecture outlined by the prominent Hungarian-American mathematician John von Neumann (1903– 1957), in collaboration with A. Burks and H. Goldstine, in 1946.
- The essence of this architecture is captured by the so-called ***random-access machine*** (***RAM***). Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed

on such machines are called ***sequential algorithms.***

Algorithm Design Techniques

An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Designing an Algorithm and Data Structures

- While the algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem may still be a challenging task.
- Sometimes, several techniques need to be combined, and there are algorithms that are hard to pinpoint as applications of the known design techniques.
- Even when a particular design technique is applicable, getting an algorithm often requires a nontrivial ingenuity on the part the algorithm designer.

Methods of Specifying an Algorithm

- Once you have designed an algorithm, you need to specify it in some fashion. In to give you an example, Euclid's algorithm is described in words (in a free and also a step-by-step form) and in pseudocode.
- These are the two options that are most widely used nowadays for specifying algorithms. Using a natural language has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult.
- ***Pseudocode*** is a mixture of a natural language and programming language like constructs. Pseudo code is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions. Surprisingly, computer scientists have never agreed on a single form of pseudo code.

Analyzing an Algorithm

- We usually want our algorithms to possess several qualities. After correctness, by far the most important is ***efficiency***.
- In fact, there are two kinds of algorithm efficiency: ***time efficiency***, indicating how fast the algorithm runs, and ***space efficiency***, indicating how much extra memory it uses.

Coding an Algorithm

- Most algorithms are destined to be ultimately implemented as computer programs.
- As a practical matter, the validity of programs is still established by testing. Testing of computer programs is an art rather than a science, but that does not mean that there is nothing in it to learn.

1.3 IMPORTANT PROBLEM TYPES

In this section, we are going to introduce the most important problem types:

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

Sorting

- The ***sorting problem*** is to rearrange the items of a given list in nondecreasing order. Of course, for this problem to be meaningful, the nature of the list items must allow such an ordering.
- As a practical matter, we usually need to sort lists of numbers, characters from an alphabet, character strings, and, most important, records similar to those maintained by schools about their students, libraries about their holdings, and companies about their employees.
- For example, we can choose to sort student records in alphabetical order of names or by student number or by student grade-point average. Such a specially chosen piece of information is called a ***key***.

Searching

- The ***searching problem*** deals with finding a given value, called a ***search key***, in a given set (or a multiset, which permits several elements to have the same value).
- There are plenty of searching algorithms to choose from. They range from the straightforward sequential search to a spectacularly efficient but limited binary search and algorithms based on representing the underlying set in a different form.

String Processing

- In recent decades, the rapid proliferation of applications dealing with nonnumerical data has intensified the interest of researchers and computing practitioners in string-handling algorithms.
- A ***string*** is a sequence of characters from an alphabet. Strings of particular interest are text strings, which comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones
- It should be pointed out, however, that string- processing algorithms have been important for computer science for a long time in conjunction with computer languages and compiling issues.

Graph Problems

One of the oldest and most interesting areas in algorithmics is graph algorithms. Informally, a ***graph*** can be thought of as a collection of points called vertices, some of which are connected by line segments called edges. (A more formal definition is given in the next section.) Graphs are an interesting subject to study, for both theoretical and practical reasons.

- Graphs can be used for modeling a wide variety of applications, including transportation, communication, social and economic networks, project scheduling, and games. The ***traveling salesman problem (TSP)*** is the problem of finding the shortest tour through *n* cities that visits every city exactly once. In addition to obvious applications involving route planning, it arises in such modern applications as circuit board and VLSI chip fabrication, X-ray crystallography, and genetic engineering.
- The ***graph-coloring problem*** seeks to assign the smallest number of colors to the vertices of a graph so that no two adjacent vertices are the same color. This problem arises in several applications, such as event scheduling: if the events are represented by vertices that are connected by an edge if and only if the corresponding events cannot be scheduled at the same time, a solution to the graph-coloring problem yields an optimal schedule.

Combinatorial Problems

From a more abstract perspective, the traveling salesman problem and the graph coloring problem are examples of ***combinatorial problems***.

These are problems that ask, explicitly or implicitly, to find a combinatorial object—such as a permutation, a combination, or a subset— that satisfies certain constraints. A desired combinatorial object may also be required to have some additional property such as a maximum value or a minimum cost

Geometric Problems

Geometric algorithms deal with geometric objects such as points, lines, and polygons. The ancient Greeks were very much interested in developing procedures (they did not call them algorithms, of course) for solving a variety of geometric problems, including problems of constructing simple geometric shapes—triangles, circles, and so on—with an unmarked ruler and a compass.

Numerical Problems

Numerical problems, another large special area of applications, are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on. The majority of such mathematical problems can be solved only approximately.

1.4 FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

The Analysis Framework

In this section, we outline a general framework for analyzing the efficiency of algorithms. We already mentioned in Section that there are two kinds of efficiency: time efficiency and space efficiency.

Time efficiency, also called ***time complexity***, indicates how fast an algorithm in question runs. ***Space efficiency***, also called ***space complexity***, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

Measuring an Input's Size

Let's start with the obvious observation that almost all algorithms run longer on larger inputs. For example, it takes longer to sort larger arrays, multiply larger matrices, and so on. Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter *n* indicating the algorithm's input size.1

- For example, it will be the size of the list for problems of sorting, searching, finding the list's smallest element, and most other problems dealing with lists.
- For the problem of evaluating a polynomial $p(x) = anxn + . . . + a0$ of degree *n*

- polynomial's degree or the number of its coefficients, which is larger by 1 than its degree. You'll see from the discussion that such a minor difference is inconsequential for the efficiency analysis.

Units for Measuring Running Time

- The next issue concerns units for measuring an algorithm's running time.. Since we are after a measure of an *algorithm*'s efficiency, we would like to have a metric that does not depend on these extraneous factors.
- One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both excessively difficult and, as we shall see, usually unnecessary.
- The thing to do is to identify the most important operation of the algorithm, called the ***basic operation***, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

Orders of Growth

- A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones. For example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid's algorithm is compared to the other two algorithms even why we should care which of them is faster and by how much.
- To find the greatest common divisor of two large numbers that the difference in algorithm efficiencies becomes both clear and important. For large values of n, it is the function's order of growth that counts: which contains values of a few functions particularly important for analysis of algorithms.
- The function growing the slowest among these is the logarithmic function. It grows so slowly, in fact, that we should expect $\log_a n = \log_a b \log_b n$

1.5 ASYMPTOTIC NOTATIONS AND BASIC EFFICIENCY CLASSES

As pointed out in the previous section, the efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency.

- To compare and rank such orders of growth, computer scientists use three notations: *O* (big oh), Ω(big omega), and *Ó* (big theta).
- In the following discussion, *t (n)* and *g(n)* can be any nonnegative functions defined on the set of natural numbers.
- In the context we are interested in, *t (n)* will be an algorithm's running time (usually indicated by its basic operation count *C(n)*), and *g(n)* will be some simple function to compare the count with.

DEFINITION A function *t (n)* is said to be in *O(g(n))*, denoted *t (n)* ε *O(g(n))*, if *t (n)* is bounded above by some constant multiple of *g(n)* for all large *n*, i.e., if there exist some positive constant *c* and some nonnegative integer *n*0 such that *t (n)* ≤ *cg(n)* for all *n* ≥ *n*0. The definition is illustrated in Figure 2.1 where, for the sake of visual clarity, *n* is extended to be a real number.

As an example, let us formally prove one of the assertions made in the introduction: 100*n* + 5 ε *O(n*2*)*. Indeed, 100*n* + 5 ≤ 100*n* + *n* (for all *n* ≥ 5) = 101*n* ≤ 101*n*2. Thus, as values of the constants *c* and *n*0 required by the definition, we can take 101 and 5, respectively.

Note that the definition gives us a lot of freedom in choosing specific values for constants *c* and *n*0. For example, we could also reason that 100*n* + 5 ≤ 100*n* + 5*n* (for all *n* ≥ 1) = 105*n* to complete the proof with *c* = 105 and *n*0 = 1.

Ω-notation

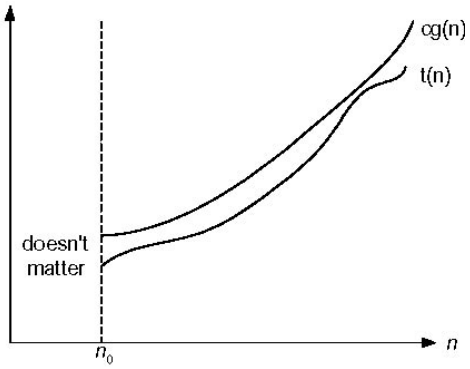
DEFINITION A function *t (n)* is said to be in *_(g(n))*, denoted *t (n)* □ *_(g(n))*, if *t (n)* is bounded below by some positive constant multiple of *g(n)* for all large *n*, if there exist some positive constant *c* and some nonnegative integer *n*0 such that

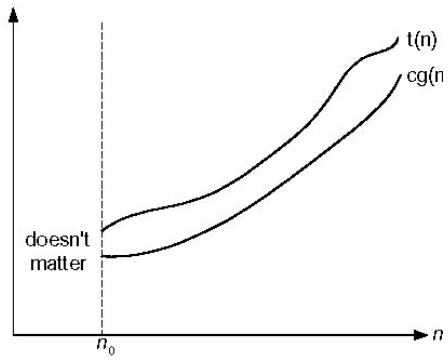
t (n) ≥ *cg(n)* for all *n* ≥ *n*0.

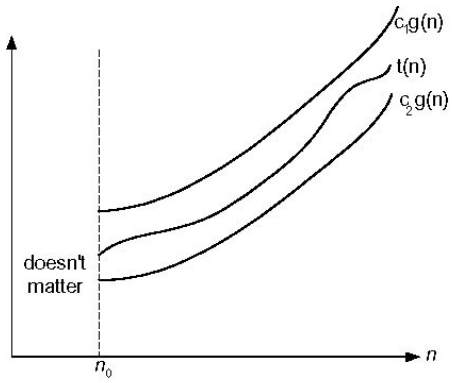
The definition is illustrated in Figure 2.2. Here is an example of the formal proof that *n*3 □ *_(n*2*)*:

*n*3 ≥ *n*2 for all *n* ≥ 0,

i.e., we can select *c* = 1 and *n*0 = 0







***Ó*-notation DEFINITION** A function *t (n)* is said to be in *_(g(n))*, denoted *t (n)* □ *_(g(n))*, if *t (n)* is bounded both above and below by some positive constant multiples of *g(n)* for all large *n*, i.e., if there exist some positive constants *c*1 and *c*2 and some nonnegative integer *n*0 such that *c2g(n)* ≤ *t (n)* ≤ *c*1*g(n)* for all *n* ≥ *n*0.

The definition is illustrated in Figure 1.3.

For example, let us prove that

1/2 *n(n* − 1*)* = *O(n*2*)*. First, we prove the right inequality (the upper bound):

THEOREM

If $t1(n) \in O(g1(n))$ and $t2(n) \in O(g2(n))$, then

$t1(n) + t2(n) \in O(\max\{g1(n), g2(n)\})$.

PROOF The proof extends to orders of growth the following simple fact about four arbitrary real numbers $a1$, $b1$, $a2$, $b2$: if $a1 \le b1$ and $a2 \le b2$, then $a1 + a2 \le 2 \max\{b1, b2\}$.

Since $t1(n) \in O(g1(n))$, there exist some positive constant $c1$ and some nonnegative integer $n1$ such that

$t1(n) \le c1g1(n)$ for all $n \ge n1$.

Similarly, since $t2(n) \in O(g2(n))$,

$t2(n) \le c2g2(n)$ for all $n \ge n2$.

Let us denote $c3 = \max\{c1, c2\}$ and consider $n \ge \max\{n1, n2\}$ so that we can use both inequalities. Adding them yields the following:

$t1(n) + t2(n) \le c1g1(n) + c2g2(n)$

$\le c3g1(n) + c3g2(n) = c3[g1(n) + g2(n)]$

$\le c32 \max\{g1(n), g2(n)\}$.

Hence, $t1(n) + t2(n) \in O(\max\{g1(n), g2(n)\})$, with the constants c and $n0$ required by the O definition being $2c3 = 2 \max\{c1, c2\}$ and $\max\{n1, n2\}$, respectively. It implies that the algorithm's overall efficiency is determined by the part with a higher order of growth, i.e., its least efficient part:

$t1(n) \in O(g1(n))$

$t2(n) \in O(g2(n))$

–

$t1(n) + t2(n) \in O(\max\{g1(n), g2(n)\})$.

For example, we can check whether an array has equal elements by the following two-part algorithm: first, sort the array by applying some known sorting algorithm; second, scan the sorted array to check its consecutive elements for equality. If, for example, a sorting algorithm used in the first part makes no more than $1/2\ n(n-1)$ comparisons (and hence is in $O(n2)$) while the second part makes no more than $n - 1$ comparisons (and hence is in $O(n)$), the efficiency of the entire algorithm will be in $O(\max\{n2, n\}) = O(n2)$.

1.6 MATHEMATICAL ANALYSIS OF NONRECURSIVE ALGORITHMS

In this section, we systematically apply the general framework outlined in Section to analyzing the time efficiency of nonrecursive algorithms. Let us start with a very simple example that demonstrates all the principal steps typically taken in analyzing such algorithms.

EXAMPLE 1

Consider the problem of finding the value of the largest element in a list of n numbers. For simplicity, we assume that the list is implemented as an array. The following is pseudocode of a standard algorithm for solving the problem.

ALGORITHM *MaxElement(A*[0..$n-1$]*)*

Determines the value of the largest element in a given array

Input: An array $A[0..n-1]$ of real numbers

Output: The value of the largest element in A

maxval ←$A[0]$

for $i \leftarrow 1$ **to** $n-1$ **do**

if $A[i] > maxval$

maxval←$A[i]$

return *maxval*

Let us denote *C(n)* the number of times this comparison is executed and try to find a formula expressing it as a function of size n. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and $n-1$, inclusive. Therefore, we get the following sum for *C(n)*:

$$C(n) = \sum_{i=1}^{n-1} 1.$$

This is an easy sum to compute because it is nothing other than 1 repeated $n-1$ times. Thus,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Here is a general plan to follow in analyzing nonrecursive algorithms.

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.

2. Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)

3. Check whether the number of times the basic operation is executed depends only on the size of

4. an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.

5. Set up a sum expressing the number of times the algorithm's basic operation is executed

6. Using standard formulas and rules of sum manipulation, either find a closedform formula for

the count or, at the very least, establish its order of growth. Before proceeding with further examples, you may want to review Appendix A, which contains a list of summation formulas and rules that are often useful

in analysis of algorithms. In particular, we use especially frequently two basic rules of sum manipulation

1.7 MATHEMATICAL ANALYSIS OF RECURSIVE ALGORITHMS

In this section, we will see how to apply the general framework for analysis of algorithms to recursive algorithms. We start with an example often used to introduce novices to the idea of a recursive algorithm.

EXAMPLE 1 Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n. Since $n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n$ for $n \geq 1$ and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm.

ALGORITHM *F(n)*

Computes *n*! recursively

Input: A nonnegative integer *n*

Output: The value of *n*!

if *n* = 0 **return** 1

else return $F(n-1) \cdot n$

For simplicity, we consider *n* itself as an indicator of this algorithm's input size (rather than the number of bits in its binary expansion). The basic operation of the algorithm is multiplication,5 whose number of executions we denote *M(n).* Since the function *F(n)* is computed according to the formula

$$F(n) = F(n-1) \cdot n \text{ for } n > 0,$$

the number of multiplications*M(n)* needed to compute it must satisfy the equality

$$M(n) = M(n-1)$$

to compute$F(n-1) + 1$

to multiply $F(n-1)$ by n for $n > 0$.

Indeed, $M(n - 1)$ multiplications are spent to compute $F(n - 1)$, and one more multiplication is needed to multiply the result by n. The last equation defines the sequence $M(n)$ that we need to find. This equation defines $M(n)$ not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely $n - 1$. Such equations are called ***recurrence relations*** or, for brevity, ***recurrences***.

Recurrence relations

- Recurrence relations play an important role not only in analysis of algorithms but also in some areas of applied mathematics. Our goal now is to solve the recurrence relation $M(n) = M(n - 1) + 1$, i.e., to find an explicit formula for $M(n)$ in terms of n only.
- Note, however, that there is not one but infinitely many sequences that satisfy this recurrence. (Can you give examples of, say, two of them?) To determine a solution uniquely, we need an ***initial condition*** that tells us the value with which the sequence starts.
- We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if $n = 0$ **return** 1.

This tells us two things:

1.First, since the calls stop when $n = 0$, the smallest value of n for which this algorithm is executed and hence $M(n)$ defined is 0. Second, by inspecting the pseudocode's exiting line, we can see that when $n = 0$, the algorithm performs no multiplications.

Therefore, the initial condition we are after is M(0) = 0. the calls stop when n = 0 no multiplications when n= 0 Thus, we succeeded in setting up the recurrence relation and initial condition or the algorithm's number of multiplications $M(n)$:

$$M(n) = M(n - 1) + 1 \text{ for } n > 0,$$

$$M(0) = 0.$$

Before we embark on a discussion of how to solve this recurrence, let us pause to reiterate an important point. We are dealing here with two recursively defined functions. The first is the factorial function $F(n)$ itself; it is defined by the recurrence

$$F(n) = F(n - 1) \cdot n \text{ for every } n > 0,$$

$$F(0) = 1.$$

2.The second is the number of multiplications$M(n)$ needed to compute $F(n)$ by the recursive

algorithm whose pseudocode was given at the beginning of the section As we just showed$M(n)$

is defined by recurrence .Though it is not difficult to "guess" the solution here (what sequence starts with 0 when $n = 0$ and increases by 1 on each step?), it will be more useful to arrive at it in a systematic fashion.

From the several techniques available for solving recurrence relations, we use what can be called the ***method of backward substitutions***. The method's idea (and the reason for the name) is immediately clear from the way it applies to solving our particular recurrence:

$M(n) = M(n - 1) + 1$ substitute $M(n - 1) = M(n - 2) + 1$

$= [M(n - 2) + 1] + 1 = M(n - 2) + 2$ substitute $M(n - 2) = M(n - 3) + 1$

$= [M(n - 3) + 1] + 2 = M(n - 3) + 3.$

After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line (what would it be?) but also a general formula for the pattern: $M(n) = M(n - i) + i.$

Strictly speaking, the correctness of this formula should be proved by mathematical induction, but it is easier to get to the solution as follows and then verify its correctness. What remains to be done is to take advantage of the initial condition given.

Since it is specified for $n = 0$, we have to substitute $i = n$ in the pattern's formula to get the ultimate result of our backward substitutions:

$M(n) = M(n - 1) + 1 = \ldots = M(n - i) + i = \ldots = M(n - n) + n = n.$

Generalizing with investigating the recursive algorithm for computing $n!$, we can now outline a general plan for investigating recursive algorithms.