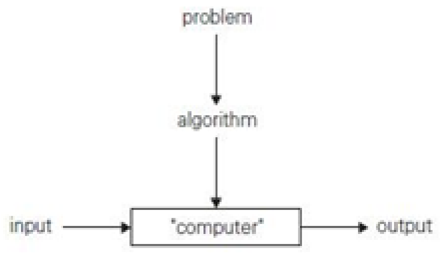
INTRODUCTION Notion of an Algorithm – Fundamentals of Algorithmic Problem Solving – Important Problem Types – Fundamentals of the Analysis of Algorithmic Efficiency –Asymptotic Notations and their properties. Analysis Framework – Empirical analysis - Mathematical analysis for Recursive and Non-recursive algorithms - Visualization	
UNIT I **PART A**	
1.	**Define Algorithm.** ***(June 06,07,May 13,17, Nov 2018)*** 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
2.	**Define order of an algorithm.** ***(June 07)*** The order of an algorithm is a standard notation of an algorithm that has been developed to represent function that bound the computing time for algorithms. The order of an algorithm is a way of defining. its efficiency. It is usually referred as Big O notation.
3.	**Write about Notion of Algorithm.** 
4.	**What are the characteristics of an algorithm?** • Input • Output • Definiteness • Finiteness • Effectiveness.
5.	**What are the different criteria used to improve the effectiveness of algorithm?** ***(June 06,07)*** • Input – Zero or more quantities are externally supplied. • Output – At least one quantity is produced • Definiteness – Each instruction is clear and unambiguous. • Finiteness – if we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps. • Effectiveness – Every instruction must be very basic.
6.	**What are the features of efficient algorithm?*(Dec 07)*** • Free of ambiguity • Efficient in execution time • Concise and compact • Completeness • Definiteness • Finiteness

7. What are the steps involved in designing and analyzing an algorithm?

- Understand the problem
- Decide on
    - Computational Device
    - Exact Vs Approximate Algorithms
    - Data Structures
    - Algorithm Design Techniques
- Design an algorithms
- Prove Correctness
- Analyze the Algorithm
- Code the Algorithm

8. What is an algorithm design techniques? ***(Dec 06)***

An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

9. What are the kinds of algorithm efficiency?

- **Time efficiency** indicates how fast the algorithm runs.
- **Space efficiency** indicates how much extra memory the algorithm needs.

10. Mention the desirable characteristics of an algorithm (*Nov 2018)*

- Simplicity
- Generality
- Optimality

11. Mention the most important problem types. ***(Dec 07,Apr 08)***

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

12. State graph-coloring problem.

The graph-coloring problem asks us to assign the smallest number of colors to vertices of a graph so that no two adjacent vertices are the same color.

13. State combinatorial problems and examples

These are the problems that ask(explicitly or implicitly) to find a combinatorial object-such as a permutation, a combination, or a subset-that satisfies certain constraints and has some desired property (maximizes a value or minimizes a cost)

14. Define basic operation in algorithm. (April/May 2018)

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.

15. Write down formula for Space Complexity Calculation? ***(Dec 13)***

S (P) = c +Sp (instance characteristics).

Where c is a constant that denotes the fixed part of the space requirements and Sp denotes the variable component.

16. Write down formula for Time Complexity Calculation?

	The Time T (P) taken by a program P is the sum of the compile time and the run time. T(p)=Compile time +Run time
17.	**How to measure an algorithm's running time? *(Dec 17)*** • Identify one or more key operations and determine the number of times these are performed .It is referred as operation count • Determine the total number of steps executed by the program. It is referred as step count
18.	**Define Asymptotic Notations. Mention the various Asymptotic Notations.*(Dec 06)*** The notation that will enable us to make meaningful statements about the time and space complexities of a program. This notation is called asymptotic notation. • Big Oh notation, • Theta notation • Omega notation • Little Oh notation.
19.	**Define Big 'O' Notation. *(June 06,May 13)*** f(n)= O(g(n)) iff positive constants c and n0 exist such that f(n)<= cg(n) for all n, $n \geq n_0$. The definition states that the function f is at most c times the function g except possibly when n is smaller than n_0.
20.	**Define Omega Notation. (June 07)** f(n)= Ω(g(n)) iff positive constants c and n0 exist such that f(n) ≥ cg(n) for all n, $n \geq n_0$. The definition states that the function f is at least c times the function g except possibly when n is smaller than n0. Here c is some positive constant. Thus g is a lower bound on the value of f for all suitably large n.
21.	**Define Theta Notation. *(Dec 14)*** f(n)= Θ(g(n)) iff positive constants c1 and c2 and an n0 exist such that c1g(n)<=f(n)<= c2g(n) for all n, n ≥ n0 . The definition states that the function f lies between c1 times the function g and c2 times the function g except possibly when n is smaller than n0. Here c1 and c2 are positive constants. Thus g is both a lower and upper bound on the value of f for all suitably large n.
22.	**List the properties of asymptotic notations. *(June 06,May 15)*** • Sum rule : O (f(n)) + O (g(n)) = O (max {f(n),g(n)}) • Transitivity rule : f(n) = O(g(n)) and g(n) = O(h(n)) then f(n) = O(h(n)) • Reflexivity: f(n) = O(f(n)) • Any constant value is equivalent to O (1). • Polynomial Rule: Let f(n) be any polynomial in n, with k being the highest exponent. f(n) = $O(n^k)$
23.	**Define best, worst and average case time complexity (Nov 2018).** The **worst-case efficiency** of an algorithm is its efficiency for the worst-case input of size n, which is an input or inputs of size n for which the algorithm runs the longest among all possible inputs of that size. The **best-case efficiency** of an algorithm is its efficiency for the best-case input of size n, which is an input or inputs for which the algorithm runs the fastest among all possible inputs of that size. The **average case efficiency** of an algorithm is its efficiency for an average case input of size n. It provides information about an algorithm behavior on a "typical" or

"random" input.

24. Write an algorithm to find the number of binary digits in the binary representation of a positive decimal integer. ***(May 15)***

```
ALGORITHM Binary(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ←1
while n > 1 do
count ←count + 1
n← └ n/2 ┘
return count
```

25. Give the Euclid's algorithm for computing gcd(m,n) ***(May 16,17) (April/May 2018)***

```
ALGORITHM Euclid(m, n)
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while n _= 0 do
r ←m mod n
m←n
n←r
return m
```

26. Compare the orders of growth of n(n-1)/2 and n2 . ***(May 16)***

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^2-n}{n^2} = \frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right) = \frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1) \in \Theta(n^2)$.

27. Define Visualization.

Algorithm visualization and can be defined as the use of images to convey some useful information about algorithms. That information can be a visual illustration of an algorithm's operation, of its performance on different kinds of inputs, or of its execution speed versus that of other algorithms for the same problem. To accomplish this goal, algorithm visualization uses graphic elements points, line segments, two- or three-dimensional bars, and so on to represent some "interesting events" in the algorithm's operation.

UNIT I

PART B

28. Describe briefly the notions of complexity of an algorithm. ***(June 07)***

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. This definition can be illustrated by a simple diagram (Figure).

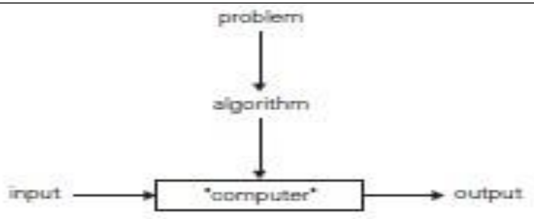


Figure: Notion of the algorithm

Important points about algorithm:

- The **nonambiguity requirement** for each step of an algorithm.
- 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**.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically **different speeds**.

Example:

- Greatest common divisor of two nonnegative, not-both-zero integers m and n
- Generating consecutive primes not exceeding any given integer $n > 1$

Example 1: GCD of two nonnegative, not-both-zero integers *m* and *n*

The greatest common divisor of two nonnegative, not-both-zero integers m and n, denoted gcd (m, n), is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero.

Method 1a: Euclid's algorithm (Structured description)
Euclid's algorithm for computing gcd (m, n)

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.
Step 2 Divide m by n and assign the value of the remainder to r.
Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

For example, gcd $(60, 24)$ can be computed as follows:

- $\gcd(m, n) = \gcd(n, m \bmod n)$
- gcd $(60, 24)$ = gcd $(24, 12)$ = gcd $(12, 0)$ = 12.

Method 1b: Euclid's algorithm (Same algorithm in pseudocode)

```
ALGORITHM Euclid (m, n)
    //Computes gcd (m, n) by Euclid's algorithm
    //Input: Two nonnegative, not-both-zero integers m and n
    //Output: Greatest common divisor of m and n
    while $n \neq 0$ do
        $r \leftarrow m \bmod n$
```

```
            m←n
            n←r
    return m
```

__Method 2: Consecutive integer checking algorithm__

Consecutive integer checking algorithm for computing gcd *(m, n)*

Step 1 Assign the value of min {*m, n*} to *t.*

Step 2 Divide *m* by *t.* If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.

Step 3 Divide *n* by *t.* If the remainder of this division is 0, return the value of *t* as the answer and stop; otherwise, proceed to Step 4.

Step 4 Decrease the value of *t* by 1. Go to Step 2.

__Method 3: Middle-school procedure__

Middle-school procedure for computing gcd*(m, n)*

Step 1 Find the prime factors of *m*.

Step 2 Find the prime factors of *n*.

Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If *p* is a common factor occurring *pm* and *pn* times in *m* and *n,* respectively, it should be repeated min{*pm*, *pn* } times.)

Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Example: Thus, for the numbers 60 and 24, we get

$60 = 2 . 2 . 3 . 5$

$24 = 2 . 2 . 2 . 3$

$\gcd(60, 24) = 2 . 2 . 3 = 12$

__Example 2: Generating consecutive primes not exceeding any given integer *n* > 1__

Sieve of Eratosthenes:

A simple algorithm for generating consecutive primes not exceeding any given integer *n* > 1. The algorithm starts by initializing a list of prime candidates with consecutive integers from 2 to *n*. Then, on its first iteration, the algorithm eliminates from the list all multiples of 2, i.e., 4, 6, and so on. Then it moves to the next item on the list, which is 3, and eliminates its multiples. No pass for number 4 is needed: since 4 itself and all its multiples are also multiples of 2, they were already eliminated on a previous pass. The next remaining number on the list, which is used on the third pass, is 5. The algorithm continues in this fashion until no more numbers can be eliminated from the list. The remaining integers of the list are the primes needed.

```
ALGORITHM Sieve (n)
//Implements the sieve of Eratosthenes
//Input: A positive integer n > 1
```

```
//Output: Array L of all prime numbers less than or equal to n
for p←2 to n do A[p]←p
for p←2 to_√n_do
    if A[p] _=0
        j ← p ∗ p
        while j ≤ n do
            A[j ]←0
            j ←j + p
i ←0
for p←2 to n do
    if A[p] _= 0
        L[i]←A[p]
        i ←i + 1
return L
```

Example: finding the list of primes not exceeding *n* = 25:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
2 3   5   7   9    11    13    15    17    19    21    23    25
2 3   5   7        11    13          17    19          23    25
2 3   5   7        11    13          17    19          23
```

For this example, no more passes are needed because they would eliminate numbers already eliminated on previous iterations of the algorithm. The remaining numbers on the list are the consecutive primes less than or equal to 25.

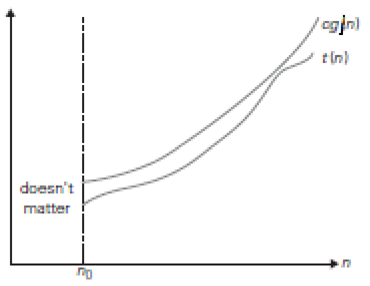
29. Explain the various asymptotic notations of an algorithm in detail (or) Explain briefly Big oh Notation, Omega Notation and Theta Notations. Give examples. Give an account of basic efficiency classes ***(Dec 07, May 17, April/may 2018, Nov 2018, Dec 06,17)***

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, we use three notations:

- O (big oh),
- *Ω* (big omega), and
- *θ* (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.

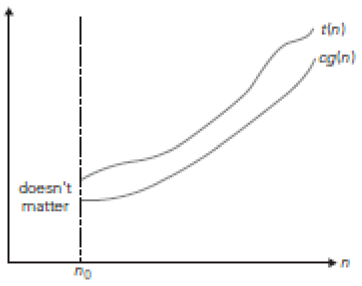
Big Oh notation ***(O)***



A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, 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

$$t(n) \le cg(n) \text{ for all } n \ge n_0.$$

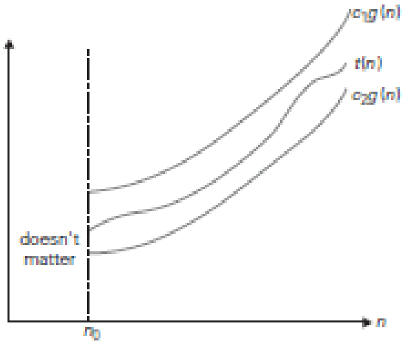
Big Omega notation (Ω)



A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega\_(g(n))$, if $t(n)$ is bounded below by some positive 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

$$t(n) \ge cg(n) \text{ for all } n \ge n_0.$$

Theta notation (θ)



A function $t(n)$ is said to be in $\theta(g(n))$, denoted $t(n) \in \theta(g(n))$, 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

$c_2g(n) \le t(n) \le c_1 g(n)$ for all $n \ge n_0$

30. **Explain the fundamentals of algorithmic problem solving. (OR) Discuss briefly**

the sequence of steps in designing and analyzing an algorithm. ***(Dec 06)***

A sequence of steps one typically goes through in designing and analyzing an algorithm (Figure) are

- Understanding the Problem
- Ascertaining the Capabilities of the Computational Device
  - Sequential algorithms Vs. parallel algorithms
  - Choosing between Exact and Approximate Problem Solving
  - Algorithm Design Techniques
- Designing an Algorithm and Data Structures
  - Methods of Specifying an Algorithm
- Proving an Algorithm's Correctness
- Analyzing an Algorithm
- Coding an algorithm

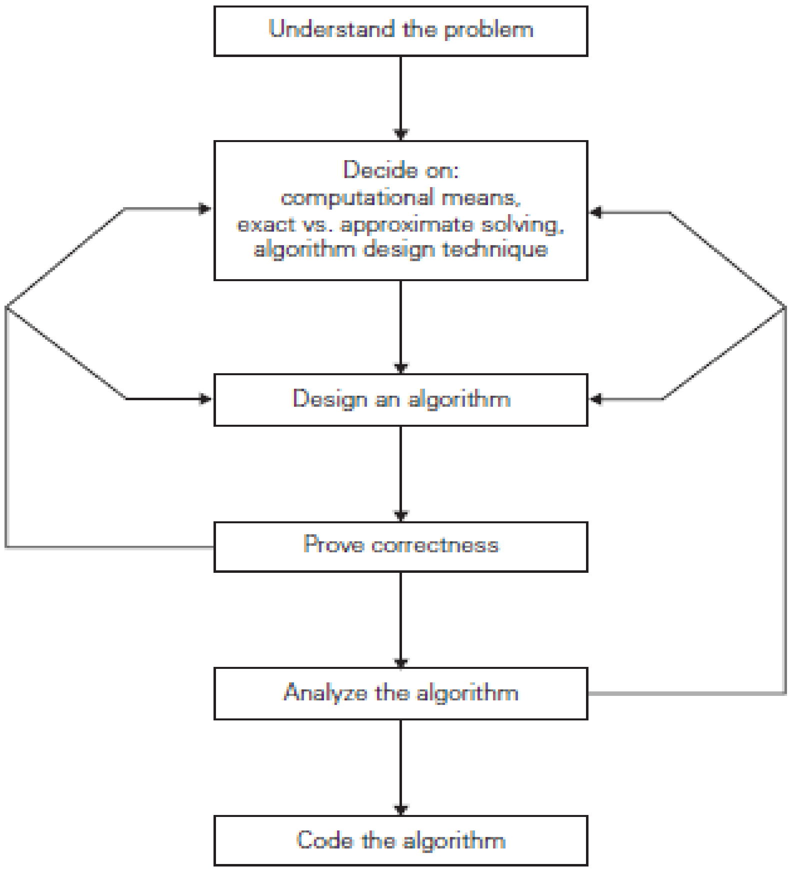


Figure: Steps in designing and analyzing algorithm

Understanding the Problem

The first thing we need to do before designing an algorithm is to **understand completely** the problem given. Read the **problem's description** carefully and **ask questions** if you have any doubts about the problem, do a few small **examples by hand**, think about **special cases**, and **ask questions again** if needed.

An input to an algorithm specifies an **instance** of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle. If you fail to do this, your algorithm may work correctly for a majority of inputs but crash on some "**boundary**" value. Remember that a correct algorithm is not one that works most of the time, but one that works correctly for ***all*** **legitimate inputs**. Do not skimp on this first step of the algorithmic problem-solving process; otherwise, you will run the risk of **unnecessary rework**.

Ascertaining the Capabilities of the Computational Device

Once we completely understand a problem, we need to ascertain the capabilities of the computational device the algorithm is intended for. They are

- Sequential algorithms Vs. parallel algorithms
- Exact algorithms Vs. approximate algorithm
- Algorithm design technique.

Sequential algorithms Vs. parallel algorithms

The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine architecture. 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. The central assumption of the RAM model does not hold for some newer computers that can execute **operations concurrently**, i.e., in parallel. Algorithms that take advantage of this capability are called parallel algorithms.

Exact algorithms Vs. approximate algorithm

The next principal decision is to choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an **exact algorithm**; in the latter case, an algorithm is called an **approximation algorithm**.

Why would one opt for an approximation algorithm?

- There are important problems that simply **cannot be solved exactly** for most of their instances. Example: extracting square roots, solving nonlinear equations, and evaluating definite integrals.
- Available algorithms for solving a problem exactly can be **unacceptably slow** because of the problem's intrinsic complexity.
- An approximation algorithm can be a **part of a more sophisticated algorithm** that solves a problem exactly.

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. The algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving.
Learning these techniques is of utmost importance for the following reasons.

- First, they provide guidance for designing algorithms for new problems, i.e.

Problems for which there is no known satisfactory algorithm.

- Second, algorithms are the cornerstone of computer science. Algorithm design techniques make it possible to classify algorithms according to an underlying design idea; therefore, they can serve as a natural way to both categorize and study algorithms.

Designing an Algorithm and Data Structures

Designing an algorithm for a particular problem may still be a challenging task because

- Some design techniques can be simply **inapplicable** to the problem in question.
- 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 of the algorithm designer.

Of course, one should pay close attention to choosing data structures appropriate for the operations performed by the algorithm. For example, the sieve of Eratosthenes would run longer if we used a linked list instead of an array in its implementation. Also some of the algorithm design techniques depend intimately on structuring or restructuring data specifying a problem's instance.

Algorithms+ Data Structures = Programs

Methods of Specifying an Algorithm

Once you have designed an algorithm, you need to specify it in some fashion. These are the three options that are most widely used nowadays for specifying algorithms.

- **Natural language**: The inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult. Nevertheless, being able to do this is an important skill that you should strive to develop in the process of learning algorithms.
- **Pseudocode***:* It is a mixture of a natural language and programming language like constructs. Pseudocode 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 pseudocode, leaving textbook authors with a need to design their own "dialects."
- **Flowchart:** A method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps. This representation technique has proved to be inconvenient for all but very simple algorithms.

Note: An algorithm's description—be it in a natural language or pseudocode—cannot be fed into an electronic computer directly. Instead, it needs to be converted into a computer program written in a particular computer language.

Proving an Algorithm's Correctness

Once an algorithm has been specified, we have to prove its ***correctness***. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time. For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd(m, n) = gcd$(n, m \bmod n)$ the simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0.

For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex. A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs. It might be worth mentioning that although tracing the algorithm's performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm's correctness conclusively. But in order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails. The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. For an approximation algorithm, we usually would like to be able to show that the error produced by the algorithm does not exceed a predefined limit.

Analyzing an Algorithm

For analyzing algorithm, the following important characteristics are considered.

- **Efficiency:** 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.
- **Simplicity:** Unlike efficiency, which can be precisely defined and investigated with mathematical rigor, simplicity, like beauty, is to a considerable degree in the eye of the beholder. For example, most people would agree that Euclid's algorithm is simpler than the middle-school procedure for computing gcd(m, n), but it is not clear whether Euclid's algorithm is simpler than the consecutive integer checking algorithm.
  Advantage:
  - Because simpler algorithms are easier to understand and easier to program;
  - Consequently, the resulting programs usually contain fewer bugs.
  - Sometimes simpler algorithms are also more efficient than more complicated alternatives.
- **Generality**: Apart from flexibility, the program should also be general. Generality means that if a program is developed for a particular task, then it should also be used for all similar tasks of the same domain. For example, if a program is developed for a particular organization, then it suit all the other similar organizations.

Coding an Algorithm

Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril and an opportunity. The validity of programs is still established by testing.

31. Explain some of the important problem types used in the design of algorithm with an example. ***(Dec 06, 07)***

The most important problem types are

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

Sorting problems

The sorting problem is to rearrange the items of a given list in nondecreasing order.

Example:

Sort lists of numbers, characters from an alphabet, character strings, and records similar to those maintained by schools about their students, libraries about their holdings, and companies about their employees. In the case of records, we need to choose a piece of information to guide sorting. 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.**

Why would we want a sorted list?

- A sorted list can be a required output of a task such as ranking Internet search results or ranking students by their GPA scores.
- Further, sorting makes searching easier : it is why dictionaries, telephone books, class lists, and so on are sorted.
- Sorting is used as an auxiliary step in several important algorithms in other areas, e.g., geometric algorithms and data compression.

Two properties of sorting algorithms are

- **Stable:** A sorting algorithm is called ***stable*** if it preserves the relative order of any two equal elements in its input. In other words, if an input list contains two equal elements in positions i and j where $i < j$, then in the sorted list they have to be in positions I and j respectively, such that $I < j$. This property can be desirable if, for example, we have a list of students sorted alphabetically and we want to sort it according to student GPA: a stable algorithm will yield a list in which students with the same GPA will still be sorted alphabetically.
- **In-place:** The second notable feature of a sorting algorithm is the amount of extra memory the algorithm requires. An algorithm is said to be ***in-place*** if it does not require extra memory, except, possibly, for a few memory units. There are important sorting algorithms that are in-place and those that are not.

Searching problems

The searching problem deals with finding a given value, called a ***search key***, in a given set. 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 more conducive to searching. It is very usefu for storing and retrieving information from large databases.

There is no single algorithm that fits all situations best. Some algorithms work faster than others but require more memory; some are very fast but applicable only to sorted arrays; and so on.

Unlike with sorting algorithms, there is no stability problem, but **different issues** arise. Specifically, in applications where the underlying data may change frequently relative to the number of searches, searching has to be considered in conjunction with two other operations: an addition to and deletion from the data set of an item. In such situations, data structures and algorithms should be chosen to strike a balance among the requirements of each operation. Also, organizing very large data sets for efficient searching poses special challenges with important implications for real-world applications.

String Processing

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; and gene sequences, which can be modeled by strings of characters from the four-character alphabet {A, C, G, T}.

String-processing algorithms have been important for computer science for a long time in conjunction with computer languages and compiling issues. One particular problem—that of searching for a given word in a text—has attracted special attention from researchers. They call it string matching. Several algorithms that exploit the special nature of this type of searching have been invented.

Graph Problems

Informally, a graph can be thought of as a collection of points called vertices, some of which are connected by line segments called edges. Graphs can be used for modeling a wide variety of applications, including transportation, communication, social and economic networks, project scheduling, and games.

Basic graph algorithms include

- Graph-traversal algorithms (how can one reach all the points in a network?),
- Shortest-path algorithms (what is the best route between two cities?), and
- Topological sorting for graphs with directed edges (is a set of courses with their prerequisites consistent or self-contradictory?).

Some graph problems are **computationally very hard**; the most well-known examples are the traveling salesman problem and the graph-coloring problem.

- The traveling salesman problem (TSP)
- The graph-coloring problem

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.

Examples of combinatorial problems: the traveling salesman problem and the graphcoloring

Combinatorial problems are the **most difficult problems** in computing, from both a theoretical and practical standpoint. Their difficulty stems from the following facts.

- First, the number of combinatorial objects typically **grows extremely fast with a problem's size**, reaching unimaginable magnitudes even for moderate-sized instances.
- Second, there are **no known algorithms** for solving most such problems exactly in an acceptable amount of time.

Geometric Problems

Geometric algorithms deal with geometric objects such as points, lines, and polygons. Of course, today people are interested in geometric algorithms with quite different applications in mind, such as computer graphics, robotics, and tomography. Examples of geometric problems are:

- The closest-pair problem: Finding the two closest points in a set of *n* points
- The convex hull of a set *S* of points is the smallest convex set containing *S*. (The "smallest" requirement means that the convex hull of *S* must be a subset of any convex set containing *S*.)

Numerical Problems

Numerical problems are problems that involve mathematical objects of continuous nature. The majority of such mathematical problems can be solved **only approximately**. Such problems typically require **manipulating real numbers**, which can be represented in a computer only approximately. Moreover, a large number of arithmetic operations performed on approximately represented numbers can lead to an accumulation of the **round-off error** to a point where it can drastically distort an output produced by a seemingly sound algorithm.

Example: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on.

Many **sophisticated algorithms** have been developed in this area, and they continue to play a critical role in many scientific and engineering applications. But now a days the computing industry has shifted its focus to **business applications**. These new applications require primarily algorithms for information storage, retrieval, transportation through networks, and presentation to users.

32. **(a)Let f(n) and g(n) be positive functions, Prove that if g(n) is Ω(f(n)) then f(n) is O(g(n)) (Nov 2018)**

Solution:

If g(n) = Ω(f(n) then there exist a constant k>0 and a constant n_0 such that all n≥ $n_{0;}$

g(n) ≥ k *f(n)

Hence:

There exist a constant k>0,and a constant n_0 such that for all n≥ $n_{0:}$ f(n)≤(1/k)*g(n)

note that since k>0, then the constant (1/k)>0

Hence:

There exist a constant c>0, namely c = (1/k) and a constant n_0 such that for all $n \geq n_0$:

$f(n) \leq c*g$ which is the definition of $f(n) = O(g(n))$

b) Briefly explain the mathematical analysis of recursive and non-recursive algorithm. ***(June 07,May 17,Nov 18)***

We systematically apply the general framework to analyzing the time efficiency of nonrecursive or iterative algorithms.

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

Step 1. Decide on a parameter indicating an **input's size**.

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

Step 3. Check whether the **number of times the basic operation is executed** depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and best-case efficiencies have to be investigated separately.

Step 4. Set up a sum expressing the number of times the algorithm's basic operation is executed.

Step 5. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count or, at the very least, **establish its order of growth**.

Example 1: Largest element in a list: 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.

```
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 ←1 to n − 1 do
        if A[i]>maxval
            maxval←A[i]
return maxval
```

Step 1: Decide on a parameter indicating an input's size:

The obvious measure of an input's size here is the number of elements in the array, i.e., *n.*

Step 2: Identify the algorithm's basic operation

The operations that are going to be executed most often are in the algorithms **for** loop. There are **two operations** in the loop's body:

Comparison operations: $A[i]> maxval$ and

Assignment operations: *maxval*←*A*[*i*].

Since the comparison is executed on each repetition of the loop and the assignment is not, we should consider the **comparison** to be the algorithm's basic operation.

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

The number of comparisons will be the same for all arrays of size *n*; therefore, in terms of this metric, there is no need to distinguish among the worst, average, and best cases here.

Step 4: Set up a sum expressing the number of times the algorithm's basic operation is executed

Let us denote *C (n)* the number of times this comparison is executed. 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)*:

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

Step 5: Establish its order of growth

This is an easy sum to compute because it is nothing other than 1 repeated n – 1 times. Thus, $C(n) = \sum_{i=1}^{n} 1 = (n-1) = \theta(n)$

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

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

2. Identify the algorithm's basic operation.

3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.

4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.

5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

Example 1: Finding Factorial

Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer *n.* Since $n! = 1 \ldots\ldots (n-1) . n = (n-1)! . n$ for $n \geq 1$ and $0! = 1$ by definition, we can compute $F(n) = F(n-1) . n$ 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) ∗ n
```

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

We consider *n* itself as an indicator of this algorithm's input size or the number of bits in its binary expansion.

Step 2. Identify the algorithm's basic operation.

The basic operation of the algorithm is multiplication, whose number of executions we denote *M(n).*

Step 3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately

Step 4: Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
Thus, we succeeded in setting up the recurrence relation and initial condition for the algorithm's number of multiplications M (n):
$M(n) = M(n - 1) + 1$ for $n > 0$,
$M(0) = 0$.
Indeed, $M(n - 1)$ multiplications are spent to compute $F(n - 1)$, and one more multiplication is needed to multiply the result by n.
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. 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 $M(0) = 0$.

Step 5. **Solve the recurrence or, at least, ascertain the order of growth of its solution.**

$$M(n) = M(n - 1) + 1 \qquad \text{substitute } M(n - 1) = M(n - 2) + 1$$
$$= [M(n - 2) + 1] + 1 = M(n - 2) + 2 \text{ substitute } M(n - 2) = M(n - 3) + 1$$
$$= [M(n - 3) + 1] + 2 = M(n - 3) + 3$$

After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line but also a general formula for the pattern:

$$M(n) = M(n - i) + i.$$

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:

$$\begin{aligned} M(n) &= M(n - 1) + 1 \\ &= \ldots \\ &= M(n - i) + i = \ldots \\ &= M(n - n) + n = n. \end{aligned}$$

33. **Explain the general framework for analyzing the efficiency of algorithm. (or) Discuss the fundamentals of analysis framework.** ***(Dec 06, 07)***

Analysis of algorithms means an investigation of an algorithm's efficiency with respect to two resources:

- Running time and
- Memory space.

This emphasis on efficiency is easy to explain.

- First, unlike such dimensions as simplicity and generality, efficiency can be studied in precise **quantitative terms**.

- Second, given the speed and memory of today's computers—that the efficiency considerations are of primary importance from a practical point of view.

THE ANALYSIS FRAMEWORK

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.

Now the amount of extra space required by an algorithm is typically not of as much concern. The time issue has not diminished quite to the same extent, however. In addition, the research experience has shown that for most problems, we can achieve much more spectacular progress in speed than in space.

Measuring an Input's Size

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**. For example,

- **Size of the list**: sorting, searching, finding the list's smallest element, and most other problems dealing with lists.
- **Polynomials degree** or **the number of its coefficients**: Evaluating a polynomial $p(x) = a_nx^n + . . . + a_0$ of degree n, it will be the polynomials degree or the number of its coefficients, which is larger by 1 than its degree.
- **Matrix order *n* or total number of elements N:** Computing the product of two $n \times n$ matrices.
- **The number *b* of bits in the *n*'s binary representation**: ($b = \log_2 n + 1$)checking primality of a positive integer n. Here, the input is just one number, and it is this number's magnitude that determines the input size.

Units for Measuring Running Time

Method 1: Standard unit of time measurement: a second, or millisecond, and so on—to measure the running time of a program implementing the algorithm. There are obvious **drawbacks** to such an approach, however:

- dependence on the speed of a particular computer,
- dependence on the quality of a program implementing the algorithm and
- dependence of the compiler used in generating the machine code

Method 2: Count the number of times each of the algorithm's operations is executed. This approach is both excessively difficult and usually unnecessary.

Method 3: Compute the number of times the basic operation is executed

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.
Rule to find basic operation: As a rule, it is usually the most time-consuming operation in the algorithm's innermost loop.

- Most **sorting algorithms** work by comparing elements (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key **comparison**.
- **Mathematical problems** typically involve some or all of the **four arithmetical operations**: addition, subtraction, multiplication, and division. Of the four, the most time-consuming operation is division, followed by multiplication and then addition and subtraction, with the last two usually considered together.

Thus, the established framework for the analysis of an algorithm's time efficiency suggests measuring it by counting the number of times the algorithm's basic operation is executed on inputs of size *n*. Here is an important application. Let

- c_{op} - Execution time of an algorithm's basic operation on a particular computer
- *C(n)*- Number of times this operation needs to be executed for this algorithm.

Then we can estimate the running time *T (n)* of a program implementing this algorithm on that computer by the formula $T(n) \approx c_{op}C(n)$.

Orders of Growth

Suppose you have analyzed two algorithms and expressed their run times in terms of the size of the input: Algorithm A takes $100n+1$ steps to solve a problem with size *n*; Algorithm B takes $n^2 + n + 1$ steps.The following table shows the run time of these algorithms for different problem sizes:

Input size	Run time of Algorithm A	Run time of Algorithm B
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$> 10^{10}$

At *n*=10, Algorithm A looks pretty bad; it takes almost 10 times longer than Algorithm B. But for *n*=100 they are about the same, and for larger values A is much better. The fundamental reason is that for large values of *n*, any function that contains an n^2 term will grow faster than a function whose leading term is *n*. The **leading term** is the term with the highest exponent.

For Algorithm A, the leading term has a large coefficient, 100, which is why B does better than A for small *n*. But regardless of the coefficients, there will always be some value of *n* where $a\,n^2 > b\,n$, for any values of *a* and *b*. The same argument applies to the non-leading terms. Even if the run time of Algorithm A were $n+1000000$, it would still be better than Algorithm B for sufficiently large *n*.

In general, we expect an algorithm with a smaller leading term to be a better algorithm for large problems, but for smaller problems, there may be a **crossover point** where another algorithm is better. The location of the crossover point depends on the details of the algorithms, the inputs, and the hardware, so it is usually ignored for purposes of algorithmic analysis. But that doesn't mean you can forget about it.

If two algorithms have the same leading order term, it is hard to say which is better; again, the answer depends on the details. So for algorithmic analysis, functions with the same leading term are considered equivalent, even if they have different coefficients.

An **order of growth** is a set of functions whose growth behavior is considered equivalent. For example, $2n$, $100n$ and $n+1$ belong to the same order of growth, which is written $O(n)$ in **Big-Oh notation** and often called **linear** because every function in the set grows linearly with n. All functions with the leading term n^2 belong to $O(n^2)$; they are called **quadratic**.

Worst-Case, Best-Case, and Average-Case Efficiencies

It is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algorithm's input. But there are many algorithms for which **running time depends not only on an input size** but also on the specifics of a particular input. Consider, as an example, sequential search that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted. Here is the algorithm's pseudocode, in which, for simplicity, a list is implemented as an array. It also assumes that the second condition $A[i] \neq K$ will not be checked if the first one, which checks that the array's index does not exceed its upper bound, fails.

```
ALGORITHM SequentialSearch(A[0..n − 1], K)
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
// or −1 if there are no matching elements
i ←0
while i < n and A[i] _= K do
        i ←i + 1
if i < n return i
else return −1
```

Clearly, the running time of this algorithm **depends not only on an input size** but also on the specifics of a particular input.

Worst-case efficiency: The worst-case efficiency of an algorithm is its efficiency for the **worst-case input** of size n, which is an input (or inputs) of size n for which the algorithm **runs the longest** among all possible inputs of that size. The way to determine the worst-case efficiency of an algorithm is to analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count $C(n)$ among all possible inputs of size n and then compute this worst-case value $Cworst(n)$. For sequential search, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n:

$C_{worst}(n) = n$.

In other words, it guarantees that for any instance of size n, the running time will not exceed $C_{worst}(n)$, its running time on the worst-case inputs.

Best-case efficiency : The best-case efficiency of an algorithm is its efficiency for the best-case input of size *n*, which is an input (or inputs) of size *n* for which the algorithm runs the fastest among all possible inputs of that size. Accordingly, we can analyze the best case efficiency as follows. First, we determine the kind of inputs for which the count *C(n)* will be the smallest among all possible inputs of size *n*. Then we ascertain the value of *C(n)* on these most convenient inputs. For example, the best-case inputs for sequential search are lists of size *n* with their first element equal to a search key; accordingly,
$C_{best}(n) = 1$

Average-case efficiency*:* It should be clear from our discussion, however, that neither the worst-case analysis nor its best-case counterpart yields the necessary information about an algorithm's behavior on a "typical" or "random" input. This is the information that the ***average-case efficiency*** seeks to provide.

To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size *n*. Let's consider again sequential search. The standard assumptions are that

- The probability of a successful search is equal to p $(0 \le p \le 1)$ and
- The probability of the first match occurring in the i^{th} position of the list is the same for every *i*.

Under these assumptions we can find the average number of key comparisons $C_{avg}(n)$ as follows.

- In the case of a **successful search**, the probability of the first match occurring in the i^{th} position of the list is *p/n* for every *i*, and the number of comparisons made by the algorithm in such a situation is obviously *i*.
- In the case of an **unsuccessful search**, the number of comparisons will be *n* with the probability of such a search being $(1-p)$. Therefore,

$$\begin{aligned}C_{avg}(n) &= [1 \cdot \tfrac{p}{n} + 2 \cdot \tfrac{p}{n} + \ldots + i \cdot \tfrac{p}{n} + \ldots + n \cdot \tfrac{p}{n}] + n \cdot (1-p)\\ &= \tfrac{p}{n}[1 + 2 + \ldots + i + \ldots + n] + n\,(1-p)\\ &= \tfrac{p}{n}\tfrac{n(n+1)}{2} + n\,(1-p)\\ &= \tfrac{p(n+1)}{2} + n\,(1-p).\end{aligned}$$

This general formula yields some quite reasonable answers. For example,

- If $p = 1$ (the search must be successful), the average number of key comparisons made by sequential search is $(n + 1)/2$; that is, the algorithm will inspect about half of the list's elements.
- If $p = 0$ (the search must be unsuccessful), the average number of key comparisons will be *n* because the algorithm will inspect all *n* elements on all such inputs.

34. a) **Discuss the general plan for analyzing the efficiency of recursive algorithm.** ***(Dec 07)***

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

b) **Write an algorithm to find the number of binary digits in the binary representation of a positive decimal integer and analyze its efficiency.** ***(Dec 07, May 16)***

Example : Number of binary digits of a decimal number

The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

```
ALGORITHM BinRec(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if n = 1 return 1
else return BinRec(_n/2_) + 1
```

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

We consider *n* itself as an indicator of this algorithm's input size or the number of bits in its binary expansion.

Step 2. Identify the algorithm's basic operation.

The basic operation of the algorithm is multiplication, whose number of executions we denote *M(n).*

Step 3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately

Step 4: Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.

Thus, we succeeded in setting up the recurrence relation and initial condition for the algorithm's number of multiplications M (n):

$M(n) = M(n-1) + 1$ for $n > 0$,

$$M(0) = 0.$$

Indeed, $M(n-1)$ multiplications are spent to compute $F(n-1)$, and one more multiplication is needed to multiply the result by *n.*

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. 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.

Step 5. **Solve the recurrence or, at least, ascertain the order of growth of its solution.**

$M(n) = M(n-1) + 1$ 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 but also a general formula for the pattern:

$M(n) = M(n-i) + i.$

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:

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

$= \dots$

$= M(n-i) + i = \dots$

$= M(n-n) + n = n.$

35. **Design a non-recursive algorithm for computing the product of two n*n matrices and also find time efficiency of algorithm.*(Dec 06)***

Given two *n* × *n* matrices *A* and *B*, compute their product *C* = *AB*. By definition, *C* is an *n* × *n* matrix whose elements are computed as the scalar (dot) products of the rows of matrix *A* and the columns of matrix *B*:

```
ALGORITHM MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
for i ←0 to n − 1 do
    for j ←0 to n − 1 do
        C[i, j ]←0.0
        for k←0 to n − 1 do
            C[i, j ]←C[i, j ]+ A[i, k] ∗ B[k, j]
return C
```

Step 1: Decide on a parameter indicating an input's size:
We measure an input's size by matrix order *n*.

Step 2: Identify the algorithm's basic operation
There are **two arithmetical operations** in the innermost loop here

- Multiplication and
- Addition

Actually, we do not have to choose between them, because on each repetition of the innermost loop **each of the two is executed** exactly once. So by counting one we automatically count the other. Still, following a well-established tradition, we consider **multiplication** as the basic operation

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

Let us set up a sum for the total number of multiplications *M(n)* executed by the algorithm. (Since this count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.)

Step 4: Set up a sum expressing the number of times the algorithm's basic operation is executed

There is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable k ranging from the lower bound 0 to the upper bound $n - 1$. Therefore, the number of multiplications made for every pair of specific values of variables *i* and *j* is

$$\sum_{i=1}^{n} 1$$

and the total number of multiplications *M(n)* is expressed by the following triple sum:

$$\text{M(n)} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

Step 5: Establish its order of growth

Now, we can compute this sum by using formula (S1) and rule (R1) given above. Starting with the innermost sum $\sum_{i=1}^{n} 1$ equal to n.

$$\text{M(n)} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

$$\text{M(n)} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n$$

$$\text{M(n)} = \sum_{i=0}^{n-1} n^2$$

$$\text{M(n)} = n^3$$

36. **Design a recursive algorithm to compute the factorial function F (n) =n! and derive the recurrence relation.*(Dec 06,17)***

Compute the factorial function *F(n)* = *n*! for an arbitrary nonnegative integer *n*. Since $n! = 1 \,.\,.\,.\,.\,.\, (n-1) \,.\, n = (n-1)! \,.\, n$ for $n \geq 1$ and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \,.\, n$ 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 $F(n - 1) * n$
```

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

We consider *n* itself as an indicator of this algorithm's input size or the number of bits in its binary expansion.

Step 2. Identify the algorithm's basic operation.

The basic operation of the algorithm is multiplication, whose number of executions we denote *M(n).*

Step 3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately

Step 4: Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.

Thus, we succeeded in setting up the recurrence relation and initial condition for the algorithm's number of multiplications M (n):

$M(n) = M(n - 1) + 1$ for $n > 0$,

$M(0) = 0.$

Indeed, $M(n - 1)$ multiplications are spent to compute $F(n - 1)$, and one more multiplication is needed to multiply the result by *n*.

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. 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 $M(0) = 0$.

Step 5. **Solve the recurrence or, at least, ascertain the order of growth of its solution.**

$$\begin{aligned} M(n) &= M(n - 1) + 1 && \text{substitute } M(n - 1) = M(n - 2) + 1 \\ &= [M(n - 2) + 1] + 1 = M(n - 2) + 2 && \text{substitute } M(n - 2) = M(n - 3) + 1 \\ &= [M(n - 3) + 1] + 2 = M(n - 3) + 3 \end{aligned}$$

After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line but also a general formula for the pattern:

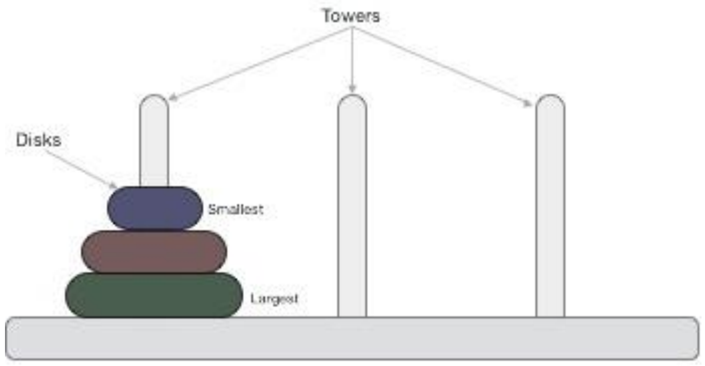
$$M(n) = M(n - i) + i.$$

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:

$$\begin{aligned} M(n) &= M(n - 1) + 1 \\ &= \ldots \\ &= M(n - i) + i = \ldots \\ &= M(n - n) + n = n. \end{aligned}$$

37. **Design a recursive algorithm to find the number of moves in tower of Hanoi problem and find the time of complexity.** ***(Dec 013)*** **(April/May 2018)**

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −



These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Solution:

We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

The steps to follow are −

Step 1 − Move n-1 disks from **source** to **aux**
Step 2 − Move n^{th} disk from **source** to **dest**
Step 3 − Move n-1 disks from **aux** to **dest**

```
START
Procedure Hanoi(disk, source, dest, aux)

  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest)     // Step 1
    move disk from source to dest          // Step 2
    Hanoi(disk - 1, aux, dest, source)     // Step 3
  END IF
END Procedure
STOP
```

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

We consider *n* itself as an indicator of this algorithm's input size or the number of bits in its binary expansion.

Step 2. Identify the algorithm's basic operation.

The basic operation of the algorithm is multiplication, whose number of executions we denote *M(n).*

Step 3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately

Step 4: Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.

Thus, we succeeded in setting up the recurrence relation and initial condition for the algorithm's number of multiplications M (n):

$M(n) = M(n-1) + 1$ for $n > 0$,

$$M(0) = 0.$$

Indeed, M(n − 1) multiplications are spent to compute *F(n − 1),* and one more multiplication is needed to multiply the result by *n.*

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. 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.

Step 5. **Solve the recurrence or, at least, ascertain the order of growth of its solution.**

$$\begin{aligned} M(n) &= M(n-1) + 1 && \text{substitute } M(n-1) = M(n-2) + 1 \\ &= [M(n-2) + 1] + 1 = M(n-2) + 2 && \text{substitute } M(n-2) = M(n-3) + 1 \\ &= [M(n-3) + 1] + 2 = M(n-3) + 3 \end{aligned}$$

After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line but also a general formula for the pattern:

$$M(n) = M(n-i) + i.$$

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:

$$\begin{aligned} M(n) &= M(n-1) + 1 \\ &= \ldots \\ &= M(n-i) + i = \ldots \\ &= M(n-n) + n = n. \end{aligned}$$

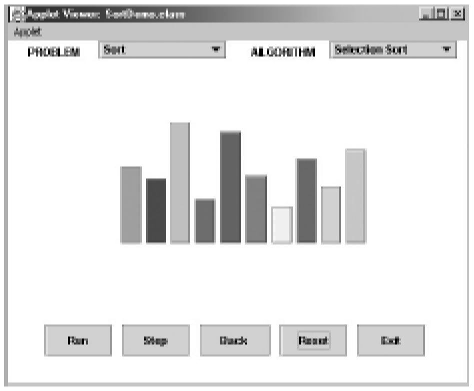
38. **Explain the algorithm visualization and its applications in detail.**

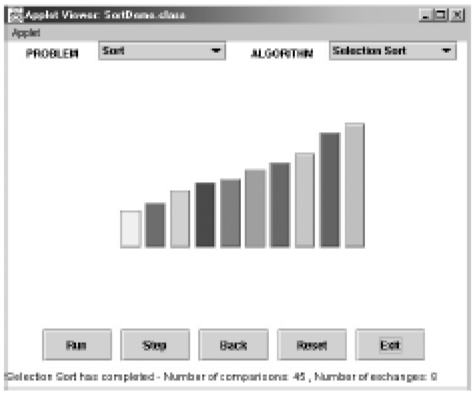
Algorithm Visualization

Three ways to study algorithms are

Mathematical analysis
Empirical analysis of algorithms,
Algorithm visualization
Empirical analysis of an algorithm: is performed by running a program implementing the algorithm on a sample of inputs and analyzing the data observed (the basic operation's count or physical running time).
Algorithm visualization: is the use of images to convey useful information about algorithms.
The two principal variations of algorithm visualization are
Static algorithm visualization shows an algorithm's progress through a series of still images and
Dynamic algorithm visualization (also called algorithm animation) shows a continuous, movie-like presentation of an algorithm's operations. Animation is an more sophisticated option and much more difficult to implement.

Initial and final screens of a typical visualization of a sorting algorithm using the bar representation:
The sorting problem lends naturally to visual presentation via vertical or horizontal bars or sticks of different heights or lengths, which need to be rearranged according to their sizes (Figure). This presentation is convenient, however, only for illustrating actions of a typical sorting algorithm on small inputs.

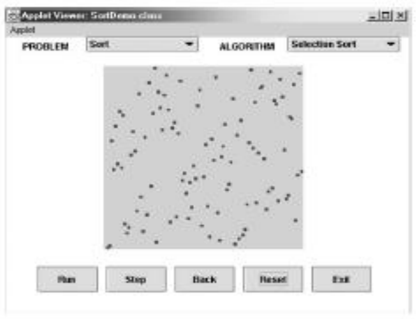




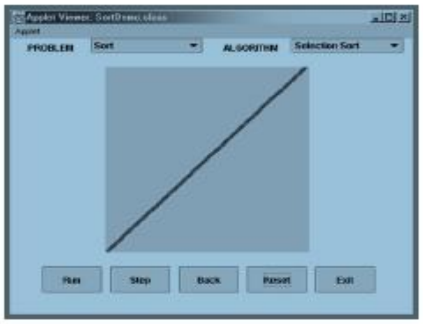
Selection Sort has completed - Number of comparisons: 45 , Number of exchanges: 9

Initial and final screens of a typical visualization of a sorting algorithm using the scatterplot representation:
For larger files, the ingenious idea of presenting data by a scatterplot of points on a coordinate plane, with the first coordinate representing an item's position in the file

and the second one representing the item's value; with such a representation, the process of sorting looks like a transformation of a "random" scatterplot of points into the points along a frame's diagonal (Figure). In addition, most sorting algorithms work by comparing and exchanging two given items at a time—an event that can be animated relatively easily.



PROBLEM Sort ALGORITHM Selection Sort



PROBLEM Sort ALGORITHM Selection Sort

Applications of algorithm visualization

There are two principal applications of algorithm visualization:
Research and
Education.

Potential benefits for researchers are based on expectations that algorithm visualization may help uncover some unknown features of algorithms. For example, one researcher used a visualization of the recursive Tower of Hanoi algorithm in which odd- and even-numbered disks were colored in two different colors. He noticed that two disks of the same color never came in direct contact during the algorithm's execution. This observation helped him in developing a better nonrecursive version of the classic algorithm.

The application of algorithm visualization to education seeks to help students learning algorithms. The available evidence of its effectiveness is decisively mixed. Although some experiments did register positive learning outcomes, others failed to do so. The increasing body of evidence indicates that creating sophisticated software systems is not going to be enough. In fact, it appears that the level of student involvement with visualization might be more important than specific features of visualization software. In some experiments, low-tech visualizations prepared by students were more effective than passive exposure to sophisticated software systems.