**UNIT I 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*** |

**1 INTRODUCTION**

* **Need for studying algorithms:** The study of algorithms is the cornerstone of computer science. It can be recognized as the core of computer science. Computer programs would not exist without algorithms.

**1.1 NOTION OF AN ALGORITHM**

**Definition:**

* 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.



**Figure 1.1:**Notion of an Algorithm.

* **Properties of algorithm:**

1. Non-ambiguity –Each step in an algorithm should be non-ambiguous. That means each instruction should be clear and precise.
2. Range of inputs.
3. The same algorithm can be represented in different ways
4. Several algorithms for solving the same problem.
   * 1. **How to write an algorithm:**

***1.Algorithm consisting of heading and body.***

*Eg:Algorithm name(P1,P2……Pn)*

***2.Then in the heading section we should write following things:***

*//Problem description*

*//input:*

*//Output:*

***3.In the body of the statement it contains various programming construct like if,for,while or some assignment statements may be written.***

***4.The compound statements should be enclosed within{and } brackets.***

***5.Single line comments are written using // as beginning of comment.***

***6.The identifier should begin by letter and not by digit.Anidentiufier can be a combination of alphanumeric string.***

***7.Using assignment operator <- an assignment statement can be given.***

***8.The array indices are stored in a [] brackets.***

***9.For input and output use Write and Read.***

***10.The conditional and looping statements are write like this:***

i)**if**(cond)**then** statement

ii)**while**(condition)**do**

{state1

…

state n}

iii)for variable🡨value1 to value n do

{state1

…

state n}

iv)**repeat**

state1

…

state n

**until**(condition)

**1.1.2 Calculating the Common Divisor**

**Three methods** for computing the Greatest Common Divisor(**gcd**) of two integers:

1. Euclid’s algorithm.
2. Consecutive integer checking algorithm.
3. Middle- School procedure.

**1.Euclid’s algorithm:gcd*(m, n)* = gcd*(n, m* mod *n****),*

* Where *m* mod *n* is the remainder of the division of *m* by *n*, until *m* mod *n* is equal to 0. The last value of *m* is also the greatest commondivisor of the initial *m* and *n*.

**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 Step2.

**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.

**Algorithm in pseudo code:**

|  |  |
| --- | --- |
| ***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*** | **For example:** gcd*(*60*,* 24*)*  *// m=60,n=24*  *While 24≠0 do*  1st iteration: r = 60 mod 24  =12; m = 24; n=12  2nd iteration: r = 24 mod 12  r = 0; m =12 ; n =0  gcd (60,24) = 12 |

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

* A common divisor cannot be greater than the smaller of the two numbers which is denoted by *t = min {m,n}.*
* Check whether‘***t***’ divides both ***m*** and***n****:* if it does, ***t*** is the answer; if it does not, then simply decrease the ***t*** by***1***and try again.Follow this method until it reaches the answer.

***Algorithm Steps:***

**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.

**Algorithm in pseudo code:**

***t ← min (m ,n)***

***if m mod t = 0 goto 3, else goto 4***

***if n mod t = 0 return t,else goto 4***

***t ← t – 1***

***goto 2***

* For example, for numbers 60 and 24, the algorithm will try first 24, then 23, and so on, until it reaches 12, where it stops.

3 **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.

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

* 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*.*

* Finding prime numbers:

1. The method used to find the prime number is known as the ***sieve of Eratosthenes.***
2. The algorithm starts byinitializing a list of prime candidates with consecutive integers from 2 to *n*.Then, on the first iteration of the algorithm eliminates from the list all multiples of 2, i.e., 4, 6, and so on.

* As an example, consider the application of the algorithm to 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

**Algorithm in pseudo code:**

*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*

***forp←2 to n do A[p]←p***

***forp←2 to√ndo***

***ifA[p] ≠ 0*** *//p hasn’t been eliminated on previous passes*

***j← p \*p***

***while j≤ n do***

***A[j ]←0 /****/mark element as eliminated*

***j←j + p****//copy the remaining elements of A to array L of the primes*

***i←0***

***forp←2 to n do***

***ifA[p] \_= 0***

***L[i]←A[p]***

***i←i + 1***

***returnL***

**1.2 FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING**

* Algorithms are the procedural solutions to problems. These solutions are not answers but specific instructions for getting answers.
* The sequence of steps in designing and analyzing an algorithm as shown in figure 1.2.

**1 Understanding the Problem**

* Thefirst step is to understand the problem statement completely before designing an algorithm. Read the problem’sdescription carefully and ask questions for clarifying the doubts about the problem.
* After carefully understanding the problem statements find out what are the necessary inputs for the solving that problem.
* The input to the algorithm is called **instance** of the problem. It is very important to **decide the range of inputs,** so that the boundary values of algorithm get fixed.
* The algorithm should work correctly for all valid inputs.

**2 Deciding on: computational means**

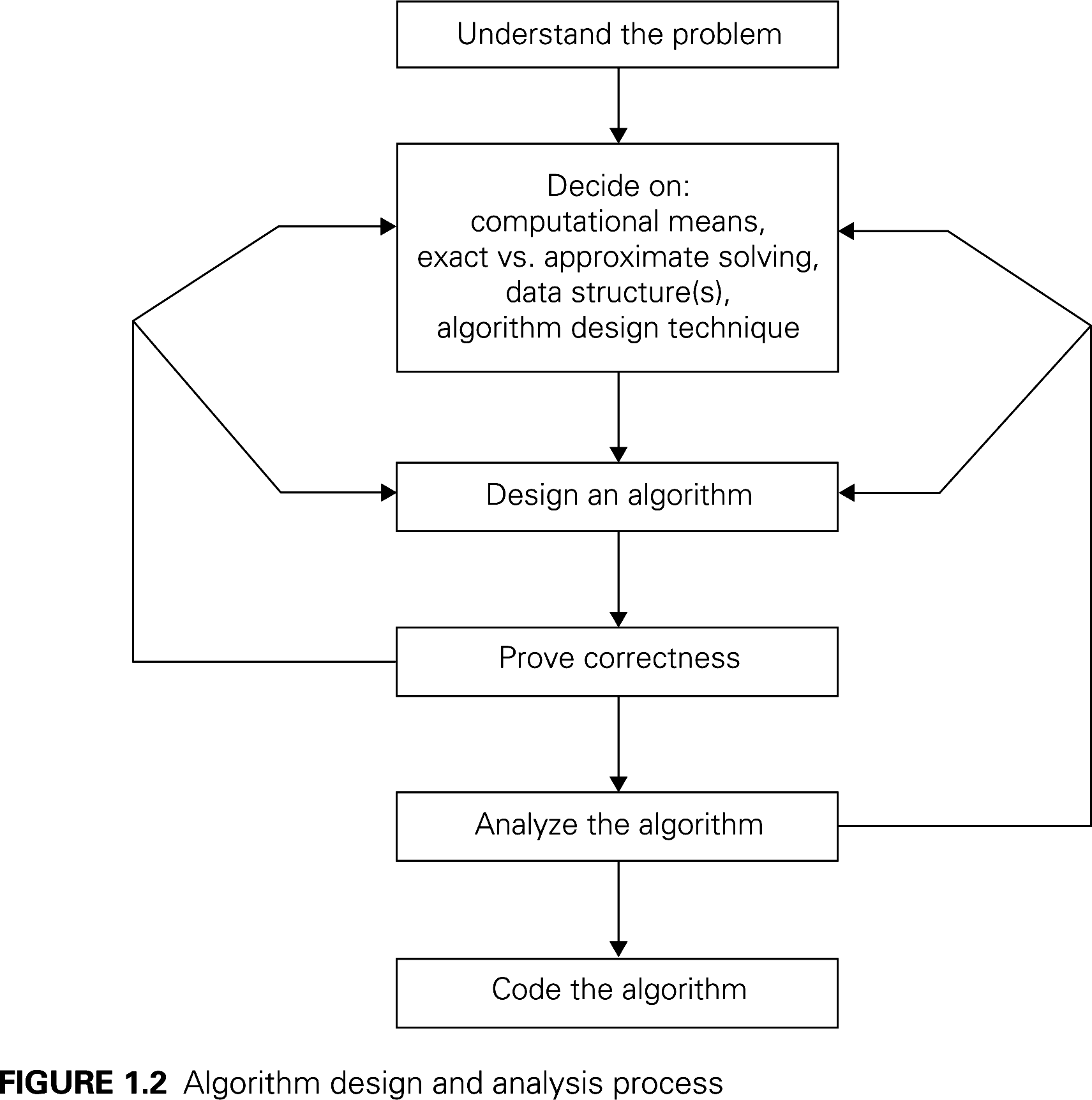
* It is necessary to know the computational capabilities of devices on which the algorithm will be running.
* ***Random-access machine*** (***RAM***) executes the instructions one after another, one operation at a time. Algorithms designed to be executed on such machines are called ***sequential algorithms***.
* 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***.

**Deciding on: exact vs. approximate solving**

* The next important decision is to decide whether the problem is to be solved exactly or approximately.
* An algorithm that can solve the problem exactly is called an **exact algorithm**.
* The algorithm that can solve the problem approximately is called an **approximation algorithm**.

**Deciding on: Algorithm design technique**

* 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.
* Algorithm design technique provides guidance for designing algorithms for new problems or problems for which there is no satisfactory algorithm. It makes it possible to classify algorithm according to underlyingdesign idea.



**Figure 1.2:** Algorithm design and analysis process.

**2.3 Design an algorithm**

* Data structure and algorithm work together and these are interdependent. Hence choice of proper data structure is required before designing the actual algorithm.

***Algorithms + Data Structures = Programs***

**2.4 Methods of Specifying an Algorithm**

* There are twodifferent ways to specifying algorithms.

1. **Pseudocode –** It is a mixture of a natural language and programming language like constructs.
2. **Flow chart-** It is a method of expressing an algorithm by a collection ofconnected geometric shapes containing descriptions of the algorithm’s steps.

**2.5 Proving an Algorithm’s Correctness**

* Correct output for every legitimate input in finite time.
* A common technique for proving correctness is to use mathematical induction because an algorithm’s iterations provide a natural sequence of stepsneeded for such proofs.
* The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms.

**2.6 Analyzing an Algorithm**

* There are two kinds of efficiency:

1. **Time efficiency**, indicating how fast the algorithm runs.
2. **Space efficiency**, indicating how much extra memory it uses.

* Characteristics of an algorithm:

1. **Simplicity:** Simplicity of an algorithm means generating sequence of instructions which are easy to understand.
2. **Generality:** There are, in fact, two issues here: generality of the problem the algorithm solves and the set of inputs it accepts.

**2.7 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 peril lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently.

Conclusion: As a rule, a good algorithm is a result of repeated effort and rework.

**1.3 IMPORTANT PROBLEM TYPES**:

**IMPORTANT PROBLEM TYPES**

**i)Sorting ii)Searching iii)String processing iv)Graph problems**

**v)Combinatorial problems vi)Geometric problems vii)Numerical problems**

**1 Sorting**

* Sorting means arranging the elements in ascending order or in descending order.
* The sorting can be done on numbers, characters, strings and records. The most important use of sorting is searching.
* The piece of information required to sort the records is called **key**. The important property of this key is that it should be unique.
* Two properties of sorting algorithms:

1. 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*
2. An algorithm is said to be *in-place* if it does not require extra memory, except, possibly, for a few memory units.

**2 Searching**

* The ***searching problem*** deals with finding a given value, called a ***search key***, in a given set.
* Some algorithms work faster but require more memory; some are very fast but applicable only to sorted arrays.
* In searching algorithms, searching has to be considered inconjunction with addition and deletion of a data item.

**3 String processing**:

* A **string**is a sequence of characters. It is mainly used in string handling algorithms.
* Most common ones are text strings, which consists of letters, numbers and special characters. Bit strings consist of zeroes and ones.
* The most important problem is the **string matching**, which is used for searching a given word in a text.

**4 Graph problems**:

* One of the interesting area in algorithmic is graph algorithms. A graph is a collection of points called vertices which are connected by line segments called edges.
* Graphs are used for modeling a wide variety of real-life applications such as transportation and communication networks.
* It includes graph traversal, shortest-path and topological sorting algorithms.
* Some graph problems are very hard; the most well-known examples are the traveling salesman problem and the graph-coloring problem.
* The ***traveling salesman problem (TSP)***, finding the shortest tour through n cities that visits every city exactly once.
* The ***graph-coloring problem*** is to assign the smallest number of colors to vertices of a graph so that no two adjacent vertices are of the same color.

**5 Combinatorial problems**

* The traveling salesman problem and the graph-coloring problem are examples of combinatorial problems.
* These are problems that ask us to find a combinatorial object such as permutation, combination or a subset- that satisfies certain constraints. These problems are difficult to solve for the following facts.
* First, the number of combinatorial objects grows extremely fast with a problem’s size.
* *Second*, there are no known algorithms for solving most such problems exactly in an acceptable amount of time.

**6 Geometric Problems**

* ***Geometric algorithms*** deal with geometric objects such as points, lines, and polygons.
* It also includes various geometric shapes such as triangles, circles etc. The applications for these algorithms are in computer graphic, robotics etc.
* The two classic problems of computational geometry: the closest-pair problem and the convex-hull problem.
* The ***closest-pair problem*** is self-explanatory: given *n* points in the plane, find the closest pair among them.
* The ***convex-hull problem*** asks to find the smallest convex polygon that would include all the points of a given set.

**7 Numerical problems**

* ***Numerical problems***, anotherspecial area of applications, are problemsthat involve mathematical objects of continuous nature: solving equations andsystems of equations, computing definite integrals, evaluating functions, and so on.
* The majority of such mathematical problems can be solved only approximately.
* Another principal difficulty stems from the fact that such problems typicallyrequire manipulating real numbers, which can be represented in a computer onlyapproximately.
* These algorithms are mainly used in scientific and engineering applications.

**1.4 FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY**

**Introduction**

* Algorithm’s efficiency is determined with respect to two resources: running time and memory space.
* The algorithm’s efficiency is represented in three notations: ***0*** (“big oh”), ***Ω*** (“big omega”) and ***θ*** (“big theta”).

**1.4.1 Analysis Framework**

* For analyzing the efficiency of algorithms there are two kinds of efficiency they aretime efficiency and space efficiency.

1. Time efficiency is measured by counting the number of times the algorithm’s basic operation is executed.
2. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.
3. Analysis framework consist of

* Measuring an input’s size
* Measuring running time
* Orders of growth (of the algorithm’s efficiency function)
* Worst-base, best-case and average-case efficiency
* **Measuring an Input’s size**

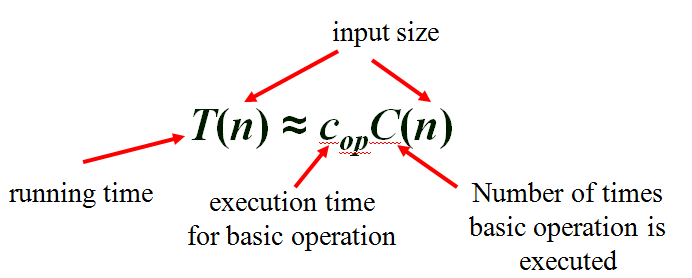
1. Input size depends on the problem. For example, it takes to sort larger arrays, multiply larger matrices and so on.
2. It is important to investigate an algorithm’s efficiency as a function of some parameter ***‘n’*** indicating the algorithm’s input size. For example, it will be the size of the list for problems of sorting, searching etc.
3. For the problem of evaluating a polynomial p (x) = an xn+ ------+ a0 of degree n, it will be the polynomial’s degree or the number of its coefficients, which is larger by one than its degree.
4. The choice of an appropriate size metric can be influenced by operations ofthe algorithm in question.
5. For e.g., in a spell-check algorithm, it examines individual characters of its input, then we measure the size by the number of characters or words.

***Note:*** measuring size of inputs for algorithms involving properties of numbers. For such algorithms, computer scientists prefer measuring size by the number b of bits in the n’s binary representation.

----------------2.1

* **Units for Measuring Running Time**

1. The standard unit of time (second or millisecond) is used to measure the running time of a program implementingthe algorithm.
2. There are obvious drawbacks to such an approach, however:
3. Dependence on the speed of a particular computer.
4. Dependence on the quality of a program implementing the algorithm.
5. Dependence on the compiler used in generating the machine code.
6. The difficulty of clocking the actual running time of the program.
7. Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*
8. **Basic operation**: the operation that contributes the most towards the running time of the algorithm.

-------2.2

* The count C (n) does not contain any information about operations that are not basic and in fact, the count itself is often computed only approximately.
* The constant *cop*is also an approximation whose reliability is not easy to assess.
* Assuming C (n) = (1/2)n(n-1), how much longer will the algorithm run if we double the input size? The answer is four times longer.
* Indeed, for all but very small values of n,

***C(n) = ½ n(n-1) = ½ n2- ½ n ≈ ½ n2***



* Here Cop is unknown, but still we got the result, the value is cancelled out in the ratio. Also, ½ the multiplicative constant is also cancelled out.

1. The efficiency analysis framework ignores the multiplicative constants of C(n) and focuses on the orders of growth of the C(n).

* **Orders of Growth**

1. Measuring the performance of an algorithm in relation with the input size n is called Orders of Growth.
2. The magnitude of the numbers in Table 2.1 has a profound significance for the analysis of algorithms.
3. The function growing the slowest among these is the logarithmic function. A logarithmic basic-operation counts to run practically directly on inputs.



1. Also note that although specific values of such a count depend, of course, on the logarithm’s base, the formula

***loga n = loga b×logb n***

1. Makes it possible to switch from one base to another, leaving the count logarithmic but with a new multiplicative constant.
2. Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.

* **Worst-case, Best-case and Average-case efficiencies**

1. The running time not only depends on the input size but also on the specifics of a particular input.
2. Example: Sequential Search(**linear search**)

*Problem:* Given a list of *n* elements and a search key *K*, find an element equal to *K*, if any.

*Algorithm:* Scan the list and compare its successive elements with *K* until either a matching element is found (*successful search*) or the list is exhausted (*unsuccessful search*)

The pseudocode is as follows.

Algorithm sequential search {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:** Returns the index of the first element of A that matches K or -1 if there is no match

***i← o***

***while i< n and A [ i ] ≠ K do***

***i← i+1***

***if i< n return i***

***else return –1***

1. In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list.So the algorithm makes the largest number of key comparisons among all possible inputs of size n; **C worst (n) = n**.

**Worst-case efficiency**

1. The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input of size n for which the algorithm runs the longest among all possible inputs of that size.
2. To analyze the algorithm determine what kind of inputs yield the largest C(n) value among all possible inputs of size n.
3. After that compute this worst-case value Cworst (n).

**Best-case efficiency**

1. The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input of size n for which the algorithm runs the fastest among all inputs of that size.
2. First, determine the kind of inputs for which the count C(n) will be the smallest among all possible inputs of size n. Then ascertain the value of C(n) on the most convenient inputs.
3. For e.g., for the searching with input size n, if the first element equals to a search key, Cbest(n) = 1.

**Average-case efficiency**

1. To analyze the algorithm’s average-case efficiency, make some assumptions about possible inputs of size n. Let us consider again sequential search.

(a) The probability of a successful search is equal to p (0 ≤ p ≤ 1), and,

(b) The probability of the first match occurring in the ithposition is same for every ‘i’.

//[sum of 1st n natural number formula]

1. This general formula yields some quite reasonable answers. For example, if *p* = 1 (i.e., successful), the average number of key comparisons made by sequential search is *(n* + 1*)/*2; that is, the algorithm will inspect, on average, about half of the list’s elements.
2. 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.

**Amortized efficiency**

1. It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure.
2. It turns out that in some situations a single operation can be expensive, but the total time for an entire sequence of *n* such operations is always significantly better than the worst-case efficiency of that single operation multiplied by *n*.
3. It is considered in algorithms for finding unions of disjoint sets.

**1.5 ASYMPTOTIC NOTATIONS AND ITS PROPERTIES**

* Asymptotic notation is the short hand way to represent the time complexity.
* 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;

**0 (big oh),**

**Ω (big omega)**

**θ (big theta)**.

* t(n) and g(n) can be any non negative functions defined on the set of natural numbers.
* t (n) is the running time of the basic operation,
* c(n) and g(n) is some function to compare the count with.

***Formal definition:***

*O***-notation**

|  |  |
| --- | --- |
| A function t(n) is said to be in O(g(n)), denoted **t(n) ∈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 n0 such that  **t(n) ≤ cg(n) for all n ≥ n0** | Figure 1.3: Big-oh notation: *t (n)* ∈ *O(g(n))*. |

**E.g.** 100n+5 ∈ O (n2)

**Proof:**  100n+ 5 ≤ 100n+n (for all n ≥ 5) = 101n ≤ 101 n2

Thus, as values of the constants c and n0 required by the definition, we can take 101 and 5 respectively. The definition says that the c and n0 can be any value. For e.g. we can also take. C = 105,and n0 = 1.

**i.e., 100n+ 5 ≤ 100n + 5n (for all n ≥ 1) = 105n**

**Ω-notation**

|  |  |
| --- | --- |
| A function *t(n)* is said to be in Ω*(g(n)),* denoted ***t(n) ∈*Ω*(g(n))****,* if *t(n)* is bounded below by some constant multiple of *g(n)* for all large *n*, i.e., if there exist some positive constant c and some nonnegative integer *n0* such that  **t(n) ≥ cg(n) for all n ≥ n0** | Figure 1.4: Big-omega notation: *t (n) ∈*Ω*(g(n))*. |

**E.g.**n3∈ Ω (n2)

**Proof:** n3≥ n2for all n ≥ 0. i.e., we can select c=1 and n0=0.

**Θ-notation**

|  |  |
| --- | --- |
| A function *t(n)* is said to be in Θ*(g(n)),* denoted *t(n) ∈*Θ*(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 constant c1 and c2 and some nonnegative integer *n0* such that  **c2 g(n) ≤ t(n) ≤ c1 g(n) for all n ≥ n0** | Figure 1.5: Big-theta notation: *t (n)* ∈Θ *(g(n))*. |

**E.g.** ½ n(n-1) ∈ θ( n2 ) .

**Proof:**

First, we prove the right inequality (the upper bound)

½ n(n-1) = ½ n2 – ½ n ≤ ½ n2for all n ≥ n0.

Second, we prove the left inequality (the lower bound)

½ n(n-1) = ½ n2 – ½ n ≥ ½ n2– ½ n½ n for all n ≥ 2 = ¼ n2.

Hence, we can select C2= ¼, C2= ½ and n0 = 2

**Useful property involving these Notations:**

**THEOREM:** If **t1 (n) ∈ O(g1(n)) and t2(n) ∈ O(g2(n)),**then**t1 (n) + t2(n) ∈ O(max{g1(n), g2(n)})**.(The analogous assertions are true for the ΘandΩnotations as well.)

**PROOF**

1. The proof extends to orders of growth the following simple fact about four arbitrary real numbers *a*1*, b*1*, a*2*, b*2: if *a*1 ≤ *b*1 and *a*2 ≤ *b*2*,* then *a*1 + *a*2 ≤ 2 max{*b*1*, b*2}*.*
2. Since *t*1*(n)* ∈*O(g*1*(n)),* there exist some positive constant *c*1 and some nonnegative integer *n*1 such that

*t*1*(n)* ≤ *c*1*g*1*(n)* for all *n* ≥ *n*1*.*

Similarly, since *t*2*(n)* ∈ *O(g*2*(n)), t*2*(n)* ≤ *c*2*g*2*(n)* for all *n* ≥ *n*2*.*

1. Let us denote *c*3 = max{*c*1*, c*2} and consider *n* ≥ max{*n*1*, n*2} so that we can use both inequalities. Adding them yields the following:

*t*1*(n)* + *t*2*(n)* ≤ *c*1*g*1*(n)* + *c*2*g*2*(n)*

≤ *c*3*g*1*(n)* + *c*3*g*2*(n)* = *c*3[*g*1*(n)* + *g*2*(n)*]

≤ *c*32 max{*g*1*(n), g*2*(n)*}*.*

1. Hence, *t*1*(n)* + *t*2*(n)* ∈*O(*max{*g*1*(n), g*2*(n)*}*)*, with the constants *c* and *n*0 required by the *O* definition being 2*c*3 = 2 max{*c*1*, c*2} and max{*n*1*, n*2}*,* respectively.

**Using Limits for Comparing Orders of Growth**



* Note that the first two cases mean that *t (n)* ∈*O(g(n)),* the last two mean that*t (n)* ∈*Ω(g(n)),* and the second case means that *t (n)* ∈*θ(g(n)).*
* The limit-based approach is often more convenient than the one based onthe definitions because it can take advantage of the powerful calculus techniques developed for computing limits, such as **L’Hopital’s rule**

and **Stirling’s formula**



**Example 1**Compare the orders of growth of *n(n* − 1*)* and *.*

*Solution:*



Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, *n(n* − 1*)* ∈*.*

**Example 2**Compare the orders of growth of log2 *n* and √*n.*

*Solution:*



Since the limit is equal to zero, log2 *n* has a smaller order of growth than√*n.*

***Little-oh notation***:



**Example 3**Compare the orders of growth of *n*! and 2*n.*



Thus, though 2*n*grows very fast, *n*! grows still faster. Symbolically that *n*!∈*Ω()*.

**Basic asymptotic efficiency classes**

|  |  |  |
| --- | --- | --- |
| **Class** | **Name** | **Comments** |
| 1 | Constant | Short of best case efficiency when its input grows the time also grows to infinity. |
| logn | Logarithmic | It cannot take into account all its input, any algorithm that does so will have at least linear running time. |
| n | Linear | Algorithms that scan a list of size n, eg., sequential search |
| nlogn | nlogn | Many divide & conquer algorithms including merge sort quick sort fall into this class |
| n2 | Quadratic | Characterizes with two embedded loops, mostly sorting and matrix operations. |
| n3 | Cubic | Efficiency of algorithms with three embedded loops, |
| 2n | Exponential | Algorithms that generate all subsets of an n-element set |
| n! | factorial | Algorithms that generate all permutations of an n-element set |

**PROPERTIES OF BIG Oh**

1. O(f(n)) + O(g(n)) = O(max{f(n), g(n)})

2. f(n) = O(g(n)) and g(n) ≤ h(n) implies f(n) = O(h(n)).

This statement shows the relaxation on tightness of asymptoticvalue of O notation.

3. Any function can be said as an order of itself. That is, f(n) =O(f(n)).

The proof of this statement trivially follows from the fact thatf(n) ≤ 1 × f(n).

1. Any constant value is equivalent to O(1). That is, C = O(1),where C is a constant.

**1.6 RECURRENCE EQUATION**

The recurrence equation is an equation that defines a sequence recursively.It is normally in following form-

T(n)=T(n-1)+n for n>0……………………………(1)

T(0)=0……………………………………………….(2)

Here the equation (1) is called recurrence relation and equation (2) is called initial condition.

**Solving Recurrence Equations**

**1.Substition method**

The substitution method is a kind of method in which a guess for the solution is made

***i).Forward substitution***

This method makes use of an initial condition in the initial term and value for the next term is generated. This process is continued until some formula is guessed. Thus in this kind of substitution method ,we use recurrence equations to generate the few terms.

***ii).Backward substitution***

In this method backward values are substituted recursively in order to derive some formula.

**2.Tree method**

The recurrence relation can also be solved using tree method. In this method, a recursion tree is built in which each node represents the cost of a single sub problem in the form of recursive function invocations.

Then we sum up the cost at each level to determine the overall cost. Thus recursion tree helps us to make a good guess of the time complexity. Let us understand this method with the help of some examples.

**3. Master’s method**

We can solve recurrence relation using a formula denoted by master’s method.

T(n)=aT(n/b)+F(n) where n>=d and d is some constant.

Then the master theorem can be stated for efficiency analysis as

**THEOREM:** If *f(n)* ∈ *θ(n d)* where *d* ≥ 0 in recurrence (2.1), then

****

**1.7 EMPIRICAL ANALYSIS**

**Definition**

* Emprical analysis means observing behavior of an algorithm for certain set of input.
* In empirical analysis actual program is written for the corresponding algorithm and with the help of some suitable input set,the algorithm is analyzed.

**General plan for empirical analysis of algorithm:**

1. Understand the purpose of experiment of given algorithm.

2. Decide the efficiency metric M.Also decides the measurement unit. For example

operations count vs. time.

3. Decide on characteristics of the input.

4. Create a program for implementing the algorithm. This program is ready for

experiment.

5. Generate a sample of input.

6. Run the algorithm for some set of input sample. Record the results obtained.

7. Analyze the resultant data.

**1. Understand the purpose of experiment of given algorithm**

* The algorithm is analyzed empirically for following reasons-
* The accuracy of an algorithm needs to be checked.
* For the same problem there can be several algorithm available and one can compare the efficiency of different algorithm for same .Or one can compare different implementations for the same problem.
* One can decide the efficiency of particular machine.
* The hypothesis for algorithm’s efficiency class can be developed.

**2.Decide the efficiency metric M.Also decide the measurement unit**

* The efficiency of an algorithm can be measured by following different methods-
* Insert a counter in the algorithm in order to count the number of times the basic operation is executed. This is a straightforward method of counting an efficiency of algorithm.

**For example: If we write a function for calculating sum of n number in an array then we can find the efficiency of that function by inserting a frequency count.The frequency count that denotes how many times the particular statement is executed.**

|  |
| --- |
| **Line 1:int sum element(int a[10],int n}**  **Line 2:{**  **Line 3:int I,sum=0;**  **Line 4:for(i=0;i<n;i++)**  **Line 5:{**  **Line 6:sum=sum+a[i];**  **Line 7:}**  **Line 8:return sum;**  **Line 9:}** |

**The execution starts from for loop.The declaration part can be neglected.Now**

|  |  |
| --- | --- |
| **Satement** | **Frequency** |
| **i=0;** | **1** |
| **i<n** | **This statement executes for (n+1)times. When condition is true i.e when i<n is true, the execution happens to be n times and the statement executes once more when i<n is false.** |
| **i++** | **n times** |
| **Sum=sum=sum+a[i]** | **n times** |
| **return sum** | **1** |
| **Total** | **(3n+3)** |

**3.Decide on characteristics of the input:**

* Even though the efficiency of basic operation is measured by frequency count or by time clocking it is necessary to consider some set of inputs for experiment.For certain sample size of input the behavior of an algorithm is to be observed. For example if we double the sample size then we can compute the ratio M(2n/M(n)) of observed metric M.
* The metric can be time or a frequency count. From this observed ratio we can identify the basic efficiency class of that algorithm. Sometimes based on random input the empirical analysis of algorithm is done. Thus range or size of the input is decided to analyze the algorithm.

**4.Generate a sample of input.**

For the decide range of input the various samples are obtained.

**5.Run the algorithm for some set of input sample. Record the results obtained.**

Using some suitable programming language the program is to be written for given algorithm and for some set of input sample the program is run. The results obtained for certain set of inputs is to be recorded. This recorded result is then presented for analysis.

**6.Analyze the resultant data.**

While analyzing the resultant data first arrange it in tabular or in graphical form.

**1.8 MATHEMATICAL ANALYSIS OF NON-RECURSIVE ALGORITHMS**

**General Plan for Analyzing the Time Efficiency of Non-recursive Algorithms**

1. Decide on a parameter (n) indicating an input’s size.
2. Identify the algorithm’s basic operation.
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, if necessary, best-case efficiencies have tobe investigated separately.

1. Set up summation for *C(n)* reflecting the number of times the algorithm’s basic operation is executed.
2. Simplify summation using standard formulas

**Two basic rules of sum manipulation**

****

and**two summation formulas**

****

**Note** that the formula=n-1 which will be used in Example 1, is a special case of formula (S1) for *l* = 1 and *u* = *n* − 1*.*

**Example 1:** Consider the problem of finding the value of **the largest element** in a list of *n* numbers.

**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

**Analysis:**

Step1: The input size (the total number of elements = n (size of the array))

Step2: Operation executed in the loop.

1. ***Comparison(***A[i] >maxval***)***
2. Assignment (maxval← A[i] )

Step3: As comparison made for each values of n there is no need to find best case, worst case and average case analysis.

Step4. Let C (*n*) denotes number of comparisons: Algorithm makes one comparison oneach execution of the loop, which is repeated for each value of the loop’s variablei within the bound between 1 and *n* – 1.

Step5:Sum for C(n)



C(n) = (n-1)-1+1 =n-1

**C(n)∈(n)**

**Example 2**Consider the ***element uniqueness problem***: check whether all the elements in a given array of *n* elements are distinct.

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

//Checks whether all the elements in a given array are distinct

//Input: An array A[0..n - 1]

//Output: Returns "true" if all the elements in A are distinct

// and "false" otherwise.

**for**i← 0 to n-2 **do**

**for** j←i: + 1 **to** n - 1 **do**

**if**A[i] = A[j] **return false**

**return true**

Step1: The **input size** is n. i.e. the total number of elements in the array.

Step2: The **basic operation** will be ***Comparison of two elements.***

Step3: The **number of element comparisons** depends not only on *n* but also on whether there are equal elements in the array.

Step4: The **worst case input** is an array for which the number of element comparisons *Cworst(n)* is the largest among all arrays of size *n.* There are **two types** of worst case inputs.

1. When there are no equal elements in the array.
2. The last two elements are equal in the array.

For such inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable *j* between its limits *i*+ 1 and *n* − 1; this is repeated for each value of the outer loop, i.e., for each value of the loop variable *i*between its limits 0 and *n* − 2*.*

*Cworst(n)=Outer loop\* Inner loop*



Ref:

C (*n*) *=*

*=*

*=-*

*=-*

=*-*

C (n) =(n-1)(n-1) –

=

=(n-1)(n-1) –

=(2(n-1)-(n-2))

=(2n-2-n+2)

=(n)=

*Cworst(n)*

**Example 3**Given two *n* × *n* matrices *A* and *B,* find the time efficiency of the definition-based algorithm for computing their product *C* = *AB.*

**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*

**Analysis:**

Step1: **input’s size** by matrix order *n*.

Step2: There are two arithmetical operations in the innermost loop here—**multiplication** and addition-consider multiplication as the **basic operation**.

Step3: Let us set up a sum for the total number of multiplications *M(n)* executed by the algorithm.

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

Step4: the total number of multiplications *M(n)* is expressed by the following triple sum:

**M(n) = *Outer loop\* Inner loop\* inner most loop(1 execution)***

=

==

=

M(n)

**Example 4** The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

**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*

**Analysis:**

Step1: The input size is n.

Step2: The basic operation is denoted by while loop. And it is each time checking whether n>1. The while loop will be executed for the number of time at which n>1 is true. It will also be executed once more than n>1 is false. But when n>1 is false the statements inside while loop won’t get executed.

Step3: The value of n is about halved on each repetition of the loop.

Step4: The exact formula for the number of times the comparison n>1 will be executed is actually

||+1

This is the number of bits in the n’s binary representation. Hence the time complexity for counting the number of bits of given number is θ). The | | indicates floor value of

**1.9 MATHEMATICAL ANALYSIS OF RECURSIVE ALGORITHMS**

* If an algorithm calls itself again and again for solving the problem then these algorithms were called Recursive algorithms.

**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**Compute the factorial function *F(n)* = *n*! for an arbitrary nonnegative integer *n.* Since *n*! = 1 *. . . . . (n* − 1*) . n* = *(n* − 1*)*! *.n*for *n* ≥ 1 and 0!= 1 by definition.

**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*

**Analysis**

Step1: The input size is n.

Step2: The basic operation is multiplication.

Step3: The recursive function call can be written as

F(n) = F(n-1)-n for n > 0,

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

M(n) = M(n - 1) + 1 for n > 0.

To compute To multiply

F(n-1) F(n-1) by n

Step4: Now we will solve recurrence using

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.

We can establish that M(n) = M (n-i)+i

Using mathematical induction prove M(n) = n

Let, n=0 then

M(n) = 0

M(0) =0= n

If we assume M(n-1) =n-1 then

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

= n-1+1 =n

M(n)=n

**The time complexity of factorial function is θ(n).**

**Example 2:** The algorithm to find 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

**Analysis**

Step1: The input size is n.

Step2: The basic operation is Division by 2

Step3: Number of additions made is A(n).

Step4: The recurrence relation is

A(n) = A(n/2) + 1 for n > 1.

A (1) = 0 -----initial condition

Step5: The standard approach to solve a recurrence is to solve if only for n=2k and then take advantage of the theorem called **smoothness rule.**

Smoothness rule claims that under very broad assumptions the order of growth observed for

*n* = 2*k*gives a correct answer about the order of growth for all values of *n.*

A(2k) = A(2k−1) + 1 for k >0,

A(20) = 0.

Now backward substitutions encounter no problems:

A(2k) = A(2k−1) + 1 substitute A(2k−1) = A(2k−2) + 1

= [A(2k−2) + 1]+ 1= A(2k−2) + 2 substitute A(2k−2) = A(2k−3) + 1

= [A(2k−3) + 1]+ 2 = A(2k−3) + 3 . . .

. . .

= A(2k−i) + i

. . .

= A(2k−k) + k.

Thus, we end up with

A(2k) = A(1) + k = k,

or, after returning to the original variable n = 2kand hence k = log2 n,

A(n) = log2 n ∈ θ(log n).

**The time complexity is θ(log n).**

**EXAMPLE 2-Tower of Hanoi**

* In this puzzle, we (or mythical monks, if you do not like to move disks) have *n* disks of different sizes that can slide onto any of three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top.
* The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.
* The problem has an elegant recursive solution, which is illustrated in Figure

To move *n>*1 disks from peg 1 to peg 3 (with peg 2 as auxiliary), we first

move recursively *n* − 1 disks from peg 1 to peg 2 (with peg 3 as auxiliary), then

move the largest disk directly from peg 1 to peg 3, and, finally, move recursively

*n*− 1 disks from peg 2 to peg 3 (using peg 1 as auxiliary).

Of course, if *n* = 1*,* we simply move the single disk directly from the source peg to the

destination peg.



The number of disks *n* is the obvious choice for the input’s size indicator, and so is

moving one disk as the algorithm’s basic operation. Clearly, the number of moves

*M(n)* depends on *n* only, and we get the following recurrence equation for it:

***M(n)* = *M(n* − 1*)* + 1+ *M(n* − 1*)* for *n >*1*.***

With the obvious initial condition *M(*1*)* = 1*,* we have the following recurrence

relation for the number of moves *M(n)*:

***M(n)* = 2*M(n* − 1*)* + 1 for *n >*1*,* (2.3)**

***M(*1*)* = 1*.***

We solve this recurrence by the same method of backward substitutions:

***M(n)* = 2*M(n* − 1*)* + 1 sub. *M(n* − 1*)* = 2*M(n* − 2*)* + 1**

**= 2[2*M(n* − 2*)* + 1]+ 1= 22*M(n* − 2*)* + 2 + 1 sub. *M(n* − 2*)* = 2*M(n* − 3*)* + 1**

**= 22[2*M(n* − 3*)* + 1]+ 2 + 1= 23*M(n* − 3*)* + 22 + 2 + 1*.***

The pattern of the first three sums on the left suggests that the next one will be

**24*M(n* − 4*)* + 23 + 22 + 2 + 1, and generally, after *i*substitutions, we get**

***M(n)* = 2*iM(n* − *i)* + 2*i*−1 + 2*i*−2 + *. . .* + 2 + 1= 2*iM(n* − *i)* + 2*i* − 1*.***

Since the initial condition is specified for *n* = 1, which is achieved for *i*= *n* − 1*,* we

get the following formula for the solution to recurrence (2.3):

***M(n)* = 2*n*−1*M(n* − *(n* − 1*))* + 2*n*−1 − 1**

**= 2*n*−1*M(*1*)* + 2*n*−1 − 1= 2*n*−1 + 2*n*−1 − 1= 2*n* − 1*.***

**1.10 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 performanceon different kinds of inputs, or of its execution speed versus that of otheralgorithms for the same problem.
* To accomplish this goal, an algorithm visualizationuses graphic elements—points, line segments, two- or three-dimensional bars, and so on—to represent some “interesting events” in the algorithm’s operation.

There are two principal variations of algorithm visualization:

* Static algorithm visualization
* Dynamic algorithm visualization, also called *algorithm animation*

**Static algorithm visualization**

* Shows an algorithm’s progress through a seriesof still images. Algorithm animation, on the other hand, shows a continuous, movie-like presentation of an algorithm’s operations.
* This presentation is convenient, however, only for illustrating actions of atypical sorting algorithm on small inputs.
* For larger files, *Sorting Out Sorting* usedthe ingenious idea of presenting data by a scatter plot of points on a coordinate plane.

**Dynamic algorithm visualization**

1. While showing the algorithm by animations ,the following features should be adopted by the developed system:
2. visualization should be consistent.
3. It should be interactive so that any ordinary user should understand it easily.
4. It should be clear and concise
5. There should be user friendliness with the developed system.
6. While developing such visualization first the developer should understand the knowledge level of the user and then accordingly the system should be designed.
7. Empasis should be on visual component rather than producing texual information.
8. Such systems should keep the user interested.
9. The symbolic and iconic representations should be incorporated.
10. The algorithmic analysis as well as comparison with different algorithms solving the same problem should be included in such animated systems.
11. The execution history should be included in such systems.

****





**FIGURE 1:** Initial and final screens of a typical visualization of a sorting algorithm using

the scatterplot representation.

**Applications Of Algorithm Visualization**:

**i)Research**

* 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 non recursive version of the classic algorithm.

**ii)Education**

* To give another example, Bentley andMcIlroy [Ben93] mentioned using an algorithm animation system in their workon improving a library implementation of a leading sorting algorithm.
* The application of algorithm visualization to education seeks to help studentslearning algorithms.

The available evidence of its effectiveness is decisively mixed.

Although some experiments did register positive learning outcomes, others failedto 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 ofstudent involvement with visualization might be more port Dynamic algorithm visualization, also called *algorithm animation.*

.