UNIT I

INTRODUCTION

Notion of an Algorithm - Fundamentals of Algorithmic Problem Solving -Important Problem Types –Algorithm Design Technique- Fundamentals of the Analysis of Algorithmic Efficiency - Asymptotic Notations and their properties-Analysis Framework – Mathematical analysis for Recursive and Non-recursive algorithms-Randomized algorithms-Las Vegas and Monte Carlo types

1.1. INTRODUCTION

Definitions

An algorithm, named after the ninth century scholar Abu Jafar Muhammad Ibu Musa Al-Khawarizmi, is defined in any one of the following ways:

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a finite step-by-step procedure to achieve a required result.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of Computation).

Algorithmics

A branch of computer science that consists of designing and analyzing computer algorithms.

The ***design*** pertain to the description of algorithm at an abstract level by means of a pseudo language, and Proof of correctness that is, the algorithm solves the given problem in all cases.

The ***analysis*** deals with performance evaluation (complexity analysis).

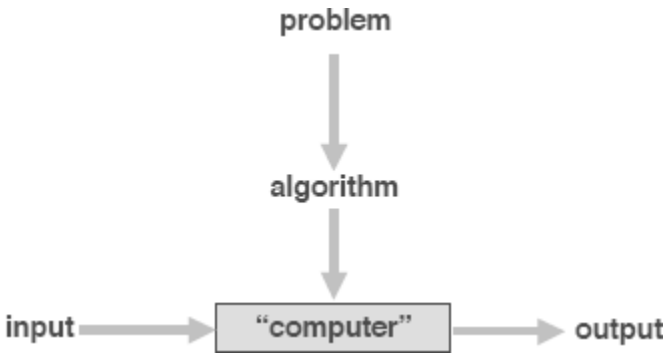
Notion of algorithm

An algorithm is a sequence of unambiguous instructions for solving a problem.

- Obtaining a required output for any legitimate input in a finite amount of time
- Procedural solutions to problems.

Computer is capable of understanding and following the instructions given.

An input specifies an ***instance*** of the problem the algorithm solves.



Motivation for Algorithm

Theoretical importance

- The study of algorithms represents the core of computer science.
- Proving the existence of a solution to problem and investigating the algorithm's properties

Practical importance

- A practitioner's toolkit of known algorithms.
- Framework for designing and analyzing algorithms for new problems.

Characteristics of an Algorithm

- ***Finiteness***

Every algorithm must terminate after a finite number of steps

- ***Definiteness***

Every step in the algorithm must rigorously and unambiguously specified

- ***Input***

Every algorithm must have a set of valid inputs which are clearly specified

- ***Output***

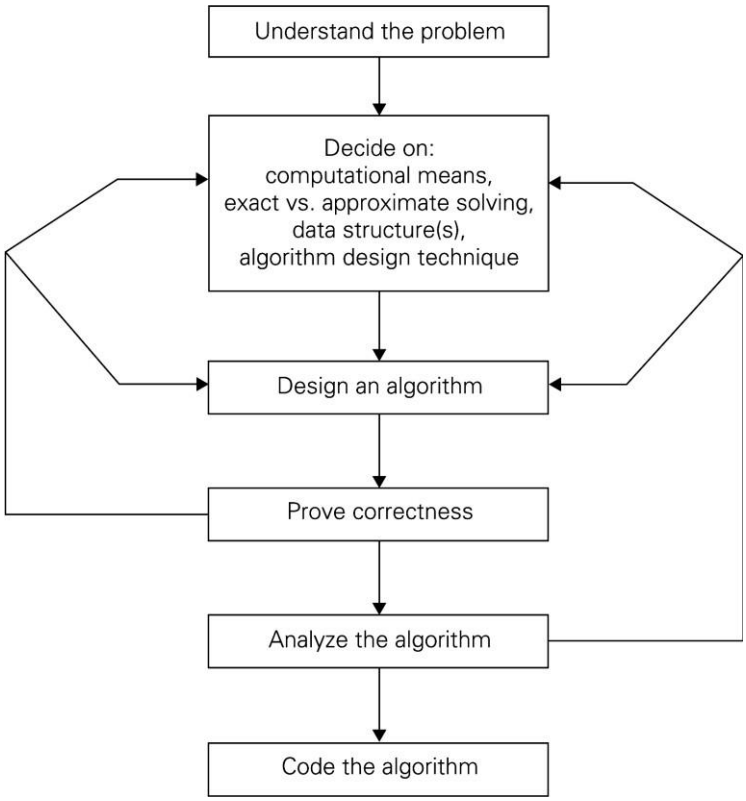
Every algorithm must produce the correct output for the given valid input

- ***Effectiveness***

The steps in the algorithm should be sufficiently simple and basic.

1.2. FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

One should go through a sequence of interrelated actions (steps) in designing and analyzing an algorithm. This is referred as ***Algorithmic Design and Analysis Process*.** A good algorithm is usually a result of repeated efforts and rework.



Understand completely the problem

- The first thing to do before designing an algorithm is to understand completely, the problem given.
- Read the problem‘s description carefully
- Do some examples by hand
- Think about special cases
- Ask questions if needed.

Ascertaining the capabilities of a computational device

- Once we completely understand a problem, we need to ascertain the capabilities of the computational device, the algorithm is intended for.
- There are two types of devices: Sequential and parallel**.**
- The algorithms designed to be executed on sequential devices are called as ***sequential(serial) algorithms***.

- The algorithms designed to be executed on parallel devices (they can execute the operations concurrently)are called as ***parallel algorithms***.

Choosing between exact and approximate problem solving

- The next thing 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***.

Deciding on appropriate data structures

- The data structure is important for both design and analysis of algorithms.
- Structuring or restructuring data specifying a problem‘s instance is important.

Algorithms + Data Structures = Programs

Algorithm design techniques

- An algorithm design technique (or ―strategy‖ or ―paradigm‖) is a general approach to solving problem algorithmically.
- The algorithm design techniques are important for the following reasons;
    a) they provide guidance for designing algorithms for new problems.
    b) they make it possible to classify algorithms according to an underlying design idea.
    c) Algorithms are the cornerstone of computer science
    d) Algorithm design techniques can serve as a natural way to categorize and study the algorithms.

Methods of specifying an algorithm

- Once we have designed an algorithm we need to specify it in some fashion.
- There are two options for specifying algorithms: ***Pseudo code*** and ***Flowchart.***
- A Pseudo code is a mixture of a natural language and a programming language-like constructs.
- A flowchart is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm‘s steps.

Proving an algorithm's correctness

- Once an algorithm has been specified, we need to prove its ***correctness.***
- That is, we have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.
- A common technique for proving the correctness is to use ***mathematic induction***
- In mathematical induction ,an algorithm's iterations provide a natural sequence of steps needed for such proofs.

Analyzing an algorithm

- After correctness, the most important is ***efficiency***.
- There are two kinds of efficiency : ***Time efficiency*** and ***Space efficiency.***
- Time efficiency indicates how fast the algorithm runs
- Space efficiency indicates how much extra memory the algorithm needs.
- Other desirable characteristics of an algorithm are ***Simplicity*** and ***Generality.***

Coding an algorithm

- An algorithm must be implemented as a computer program with test and debugging for its validation.
- We must make sure to provide verifications on the range and validity of the input.

1.3. IMPORTANT PROBLEM TYPES

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

SORTING

The sorting problem asks us to ***rearrange the items*** of a given list in some order (say, in an ascending order).

Examples

Lists of numbers

Characters from an alphabet

Character strings

Records by schools about their students

Employee details by companies

Sorting makes searching easier.

Dictionaries

Telephone books

Class lists and so on

To sort records, some information called a ***key*** (e.g., name or ID number) is chosen.

The main objectives involved in the design of sorting algorithms are

Minimum number of exchanges

Large volume of data block movement

The important criteria for the selection of sorting algorithms is as follows :

Programming time of the sorting algorithm

Execution time of the program

Memory space needed for the programming environment

Properties of sorting algorithms- ***Stable*** and ***In-place***

Stable: A sorting algorithm is called stable, if it preserves the relative order of any two equal elements in its input. That is, 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‘.

In-place : An algorithm is said to be in place ,if it does not require extra memory.

Types of sorting

There are two major classifications of sorting algorithms: Internal sorting and External sorting

Internal sorting

The key principle of internal sorting is that all the data items to be sorted are retained in the main memory and random access into this memory space can be effectively used to sort the data items. The various internal sorting methods algorithms are

a) Bubble sort
b) Selection sort
c) Shell sort
d) Insertion sort
e) Quick sort
f) Heap sort

External sorting

The idea behind the external sorting is to move the data from secondary storage to main memory in blocks for sorting them.

The most commonly used external sorting method is the merge sort.

SEARCHING

The searching problem deals with finding a given value, called a ***search key***, in a given set.

Searching algorithms

a) Sequential or Linear search
b) Binary search
c) Hashing
d) Tree-based searches
    i. Binary search tree
    ii. Balanced search trees

Searching should be considered in conjunction with the ―addition to‖ and ―deletion from‖ the dataset of an item. Searching algorithms are used for storing and retrieving information from large databases.

STRING PROCESSING

A string is a sequence of characters from an alphabet.

Examples :

Text strings, which comprise letters, numbers, special characters

Bit strings, which comprise zeros and ones

Gene sequences, which are strings of characters from A, C, G and T.

Pattern matching or String matching – The process of searching for an occurrence of a given word or a pattern in a text.

The algorithms used for String matching are

(i) String matching algorithm by Knuth-Morris-Pratt

- compares characters of a pattern from left to right.

(ii) String matching algorithm by Boyer – Moore

- compare characters of a pattern from right to left.

GRAPH PROBLEMS

A graph is a collection of points (vertices) which are connected by lines (edges).

Graphs can be used for modeling a wide variety of real-life applications, which includes

Transportation

Communication networks

Project scheduling

Games

Web‘s diameter (Maximum no. of links one needs to follow to reach one web page from another)

Basic graph Algorithms

a) Graph traversal - How can one visit all points in a network?
    i. Breadth first search
    ii. Depth first search
b) Shortest-path problem - What is the best route between two cities?
    i. Floyd‘s algorithm
    ii. Dijkstra‘s algorithm
    iii. Traveling salesman problem
c) Topological sorting problem - Listing the vertices of a digraph
d) Minimum spanning tree Algorithms - A minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

1. Prim‘s Algorithm
2. Kruskal‘s Algorithm

e) Graph coloring – Assign the smallest number of colors to vertices of a graph
   1. such that no two vertices are of the same color.

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 constrains and has some desired property. (e.g., maximize a value or minimize a cost).

Combinatorial problems are the most difficult problems in computing from both the theoretical and the practical point of view. There are two reasons for this difficulty :

(i) The number of combinatorial objects typically grows extremely fast with a problem‘s size

(ii) There are no known algorithms for solving such problems

Examples

Traveling salesman problem

Knapsack problem

Assignment problem

Graph coloring problem

Algorithms

a) Generating permutations
b) Generating subsets
c) Computing a binomial coefficient
d) Brute-force approaches based on exhaustive search
e) Algorithms based on dynamic programming and memory function method.

GEOMETRIC PROBLEMS

Geometric algorithms deal with geometric objects such as points, lines, and polygons.

Some of the interesting geometric problems

a) Closest-pair problem: Find the closest pair among given n points in the plane

b) Convex hull problem: Find the smallest convex polygon that would include all the points of a given set

c) Dihedral angle problem: Calculate the dihedral (torsional) angle for four points to decide whether they all lie in the same plane.

NUMERICAL PROBLEMS

- Numerical problems are those that involve mathematical objects of continuous nature.
- Most of the mathematical problems can be solved only approximately.
- These problems perform arithmetic operations on real numbers, which can be represented in a computer only approximately.
- Many sophisticated algorithms were developed for many scientific and engineering applications.
- Now the computing industry in focusing in business applications. As a result, numerical analysis has lost its dominating position.

Examples

Solving equations, and systems of equations

Computing definite integrals: trapezoidal rule

Evaluating functions.

Taylor polynomial

Newton‘s algorithm for computing square roots.

1.4. FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY - ANALYSIS FRAMEWORK

Analysis of algorithms means to investigate an algorithm‘s efficiency with respect to running time and memory space (i.e) : ***Time efficiency*** and ***Space efficiency.*** Time efficiency indicates how fast the algorithm runs; Space efficiency indicates how much extra memory the algorithm requires.

1. Measuring an input‘s size
2. Units for Measuring running time
3. Orders of growth
4. Worst-case, best-case and average-case efficiency

1.4.1 Measuring an Input Size

Efficiency is defined as a function of input size. Input size depends on the problem.

Eg 1 : what is the input size of the problem of sorting n numbers?

Eg 2 : what is the input size of adding two n by n matrices?

All algorithms run longer on larger inputs. For eg, 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 eg, it will be the size of the list for problems of sorting, searching etc.

Size of inputs for algorithms involving properties of numbers is often expressed by the number b of bits in the n‘s binary representation:

$$b = [\log_2 n]+1$$

1.4.2 Units for Measuring running time

We measure the running time using standard unit of time measurements, such as second, millisecond, microsecond and so on.

The running time depends on the following factors

- the speed of the computer
- the quality of a program
- the complier used

Since we need to measure an algorithm‘s efficiency, we should have a metric that does not depend on these factors.

One possible approach is to count the number of times, the algorithms operations are executed. Bit this approach is difficult and un-necessary.

The best approach is to identify the most important operation of the algorithm, called ***the basic operation***, the operation that contributes the most to the total running time and ***compute the number of times, the basic operation is executed.***

Basic Operation: The basic operation is usually the most time-consuming operation in the algorithm‘s inner-most loop.

For eg., (i) In sorting algorithms,the basic operation is a key comparison.

(ii) In matrix multiplication , the basic operation is multiplication and addition.

Computing the number of times, the basic operation is executed : Let c_{op} be the execution time of an algorithms basic operation on a particular computer, and let C(n) be the number of times this operation is run on input n.Then we can estimate the running time T(n) of a program by the formula

$$T(n) \approx c_{op} C(n)$$

Suppose we observe that some algorithm, A, makes C(n) basic operations for an input of size n, where C(n) is given by

$$C(n) = \frac{1}{2} n (n-1)$$

How much longer would the algorithm require for 2 times its input i.e: $T(2n) \approx ?$

The answer is about 4 times longer.

$$C(n) = \frac{1}{2} n (n-1) = \frac{1}{2} n^2 - \frac{1}{2} n \approx \frac{1}{2} n^2$$

and therefore

$$\frac{T(2n)}{T(n)} \approx \frac{copC(2n)}{copC(n)} \approx 4$$

1.4.3 Orders of growth

A difference in running times on small number of inputs does not distinguish between the efficient and inefficient algorithms.It is only the large number of inputs that shows the real difference.. For large values of n, it is the function‘s order of growth that counts.

The table below contains values of a few functions that are particularly important for the analysis of algorithms.

Exponential Growth Functions

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

1.4.4 Worst-case, best-case and average-case efficiency

Worst-case

- The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, for which the algorithm runs the longest among all possible inputs of size n.
- The way to determine the worst-case efficiency of an algorithm is to see what kind of inputs yield the largest value of the operation‘s basic count C(n). It is indicated by C_{worst} (n) .

Best – case

- The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, for which the algorithm runs the fastest among all possible inputs of size n.
- The way to determine the worst-case efficiency of an algorithm is to see what kind of inputs yield the smallest value of the operation‘s basic count C(n). It is indicated by C_{best} (n) .

Average-case

- Neither the worst-case nor the best-case analysis yields the necessary information about an algorithm‘s behavior on a ―typical‖ or ―random input‖.For these kind of inputs,we need to determine the average-case.
- To analyze the average-case efficiency,we need to make some assumptions about the possible inputs of size n.

Amortized efficiency

- Another type of efficiency is called amortized efficiency

- It applies not to a single run of an algorithm, but rather to a sequence of operations performed on the same data structure.
- (i.e) it turns out to be expensive for a single operation but the total time for an entire set of operations would be better .

Example

```
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: Returns the index of the first element of 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
```

For the above example, we will show the various efficiencies:

(i) worst-case efficiency : $C_{worst}(n) = n$

(ii) best-case efficiency : $C_{best}(n) = 1$

(iii) average-case efficiency : $C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p)$,

Where probability of a successful search = p

p = 1, for successful search and $C_{best}(n) = \frac{(n+1)}{2}$

p = 0, for unsuccessful search and $C_{best}(n) = n$

1.5. ASYMPTOTIC NOTATIONS

Asymptotic notations are used to compare the orders of growth of an algorithm's basic operation count, that ignores constant factors and small input sizes.

Three notations are used

- O – notation (Big-Oh)
- Ω - notation (Big-omega)
- Θ - notation (Big-Theta)

O - Notation

O (g(n)) is the set of all functions with a smaller or same order of growth as g(n).

Formal Definition

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 non-negative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$

Examples

$10n^2 \in O(n^2)$

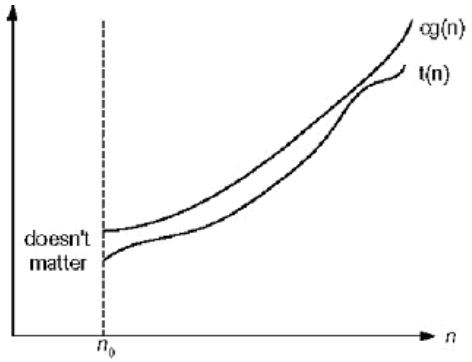
$10n^2 + 2n \in O(n^2)$

$100n + 5 \in O(n^2)$

$5n+20 \in O(n)$

$n^3 \notin O(n^2)$

Diagram



Big-oh notation: $t(n) \in O(g(n))$

Ω - Notation

Ω(g(n)) is the set of all functions with a larger or same order of growth as g(n).

Formal definition

A function t(n) is said to be in Ω(g(n)), denoted $t(n) \in \Omega(g(n))$, 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 n_0 such that

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

Examples

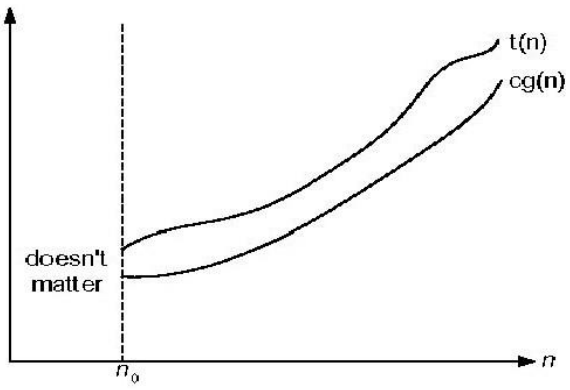
$10n^2 \in \Omega(n^2)$

$10n^2 + 2n \in \Omega(n^2)$

$10n^3 \in \Omega(n^2)$

$100n + 5 \notin \Omega(n^2)$

Diagram



Big-omega notation: $t(n) \in \Omega(g(n))$

Θ - Notation

Θ(g(n)) is the set of all functions that have the same order of growth as g(n).

Formal definition

A function t(n) is said to be in Θ (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 constant c_1 and c_2 and some nonnegative integer n_0 such that

$c_2\, g(n) \leq t(n) \leq c_1\, g(n)$ for all $n \geq n_0$

Examples

$10n^2 \in \Theta(n^2)$

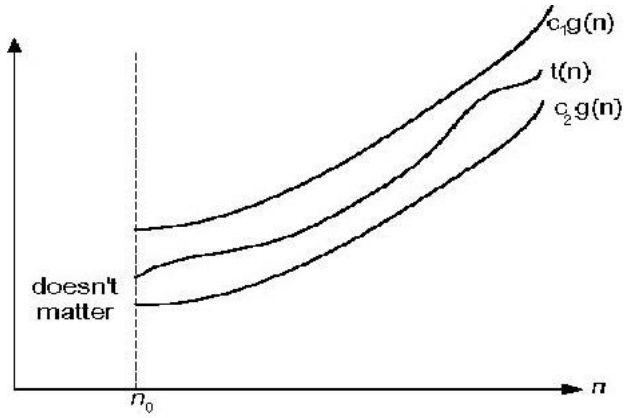
$10n^2 + 2n \in \Theta(n^2)$

$\frac{1}{2} n(n-1) \in \Theta(n^2)$

$100n + 5 \notin \Theta(n^2)$

$10n^3 \notin \Theta(n^2)$

Diagram



Big-theta notation: $t(n) \in \Theta(g(n))$

Properties of Asymptotic Notations

1.$t(n) \in O(t(n))$

2.$t(n) \in O(g(n))$ iff $g(n) \in \Omega(t(n))$

3.If $t(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $t(n) \in O(h(n))$

4. If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$

Note : The analogous assertions are true for the Ω-notation and Θ-notation.

Theorem and its Proof

Theorem **:** If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

Proof **:** Since $t_1(n) \in O(g_1(n))$, there exists some constant c_1 and some non-negative integer n_1 such that

$$t_1(n) \leq c_1 g_1(n), \quad \text{for all } n \geq n_1 \qquad \rightarrow \text{Eqn 1}$$

Similarly , since $t_2(n) \in O(g_2(n))$, there exists some constant c_2 and some non- negative integer n_2 such that

$$t_2(n) \leq c_2 g_2(n), \quad \text{for all } n \geq n_2 \qquad \rightarrow \text{Eqn 2}$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \geq \max\{n_1, n_2\}$ so that we can use both the inequalities.

Adding the two inequalities, we get

$$t_1(n) + t_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$$
$$\leq c_3 g_1(n) + c_3 g_2(n) = c_3 [g_1(n) + g_2(n)]$$
$$\leq c_3\, 2 \max\{g_1(n), g_2(n)\}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 being $2c_3 = 2 \max\{c_1, c_2\}$ and $\max(n_1, n_2)$ respectively.

Using Limits for Comparing Orders of Growth

There are three cases for computing the limit of the ratio of two functions.

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} =$$

- 0 implies that *t(n)* has a smaller order of growth than *g(n)*
- *c* implies that *t(n)* has the same order of growth as *g(n)*
- ∞ implies that *t(n)* has a larger order of growth than *g(n)*

The first two cases mean that $t(n) \in O(g(n))$, the last two cases mean that $t(n) \in \Omega(g(n))$ and the second case means that $t(n) \in \Theta(g(n))$.

L'Hôpital's rule

If $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$ and the derivatives f', g' exist, then

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

Stirling's formula

$$n! \approx \sqrt{2\Pi n}\,(\tfrac{n}{e})^n \text{ for large values of n.}$$

Example 1

Compare orders of growth of $\frac{1}{2}n(n\text{-}1)$ and n^2.

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

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

Example 2

Compare orders of growth of $\log_2 n$ and $\sqrt{n}$.

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}^2} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{\sqrt{n}}{n} = 0$$

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than $\sqrt{n}$.

Example 3

Compare orders of growth of n! and 2^n.

Taking the advantage of Stirling's formula,

$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{\sqrt{2\Pi n}\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty} \sqrt{2\Pi n}\,\frac{n^n}{2^n e^n} = \lim_{n\to\infty} \sqrt{2\Pi n}\cdot\left(\frac{n}{2e}\right)^n = \infty.$$

Thus though 2^n grows very fast, n! grows still faster. Hence it is written symbolically, that $n! \in \Omega(2^n)$.

Basic asymptotic efficiency classes

Class	Name	Comments
1	constant	For best case efficiencies
log n	logarithmic	For problems in which the size of the inputs are cut by a constant factor on each iteration of the algorithm
n	linear	For algorithms that scan a list of size _n‘
n log n	n-log-n	For many divide and conquer algorithms
n^2	quadratic	For algorithms with two embedded loops
n^3	cubic	For algorithms with three embedded loops
2^n	exponential	For algorithms that generate subsets of an n-element set
n!	factorial	For algorithms that generate all permutations of an n-element set

1.6. MATHEMATICAL ANALYSIS OF NON-RECURSIVE ALGORITHMS

Non-recursive algorithms are executed only once to solve the problem.

Examples

- Largest element in a list of numbers
- Element uniqueness problem
- Matrix operations: transpose, addition and multiplication
- Digits in binary representation

Steps or General Plan for analyzing the efficiency of Non-recursive algorithms

1. Decide on parameter (or parameters) indicating an input‘s size
2. Identify algorithm‘s basic operation
3. Check whether the number of times the basic operation is executed, depends only on the input size n. If it also depends on some additional property, investigate worst, average, and best case efficiency separately.

4. Set up a sum for expressing the number of times the algorithm‘s basic operation is executed.
5. Using standard formulas and rules of summation compute a closed-form formula for the count or compute its order of growth.

Example 1: Consider the problem of finding the value of the largest(maximum) 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

Input size : n, the number of elements in the array

Basic Operation : Comparison . It is executed on each repetition of the loop

Formula for the basic operation count: Sum is simply 1 repeated by n-1 times.

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

Example 2 : Consider the element uniqueness problem: check whether all the elements in a given array 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
```

Analysis

Input Size : n ,the number of elements in the array

Basic Operation : Comparison in the innermost loop

Formula for the basic operation count :

Depends also on whether there are equal elements in the array and, if there are, which positions they occupy.

In worst case, the algorithm needs to compare all n(n-1)/2 distinct pairs.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1) - (i+1) + 1 = \sum_{i=0}^{n-2} (n-1-i)$$

$$= (n-1) + (n-2) + \ldots\ldots\ldots\ldots\ldots + 1 \approx \frac{(n-1)n}{2}$$

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

Example 3 : Consider the multiplication of two matrices

```
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-by-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

Input Size : n , the matrix order

Basic Operation : Two operations - Multiplication and Addition

Formula for the basic operation count:

One multiplication is executed on each repetition of the innermost loop (***k***) and we need to find the total number of multiplications made for all pairs of ***i*** and ***j***

The algorithm computes n^2 elements of the product matrix. Each element is computed as the scalar (dot) product of an n-element row of A and an n- element column of B, which takes n multiplications.

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

$$= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n*n = n * n * n = \Theta(n^3)$$

Example 4 : Consider the problem of finding the number of binary digits in the binary representation of a positive decimal number.

```
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

The comparison, n > 1, decides whether the while loop will be executed.

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

Number of times the comparison is executed = number of repetitions of the loop's body + 1.

Hence we have , $C(n) = \log_2 n + 1 = \Theta(\log n)$.

1.7. MATHEMATICAL ANALYSIS OF RECURSIVE ALGORITHMS

Recursive algorithms are invoked (makes reference to) itself repeatedly until a certain condition matches.

Examples

- Computing factorial function

- Tower of Hanoi puzzle
- Digits in binary representation

Steps or General Plan for analyzing the efficiency of Recursive algorithms

1. Decide on parameter (or parameters) indicating an input's size
2. Identify algorithm's basic operation
3. Check whether the number of times the basic operation is executed, depends only on the input size n. If it also depends on some additional property, investigate worst, average, and best case efficiency separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the algorithm's basic operation is executed.
5. Solve the recurrence relation using backward substitution or compute its order of growth.

Example 1 : Computing the factorial of a given number.

$F(n) = F(n-1) * n$ for $n \geq 1$ and $F(0) = 1$

```
Algorithm F(n)
    //Compute n! recursively
    //Input: A nonnegative integer n
    //Output: The value of n!
    if n = 0 return 1
    else return F(n-1) * n
```

Analysis

Input Size : n

Basic Operation : Multiplication

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

Initial Condition $M(0) = 0$

Solving by backward substitution, we get

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

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

$= (M(n-3) + 1) + 2 = M(n-3) + 3$

…

= M(n-i) + i

Substituting for i = n, we get

= M(n-n) + n

= M(0) + n

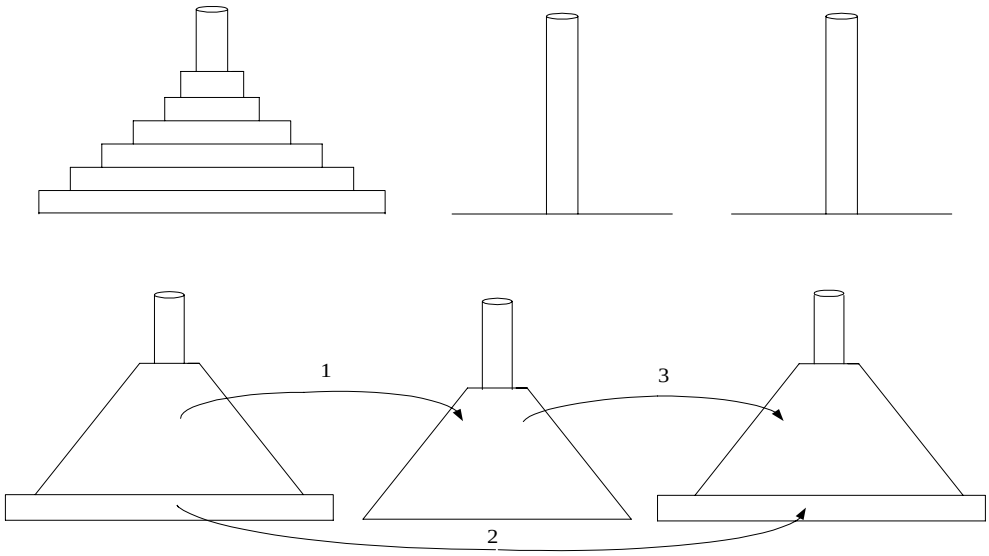
= 0 + n

= n

Example 2 : Consider the Tower of Hanoi problem.

Given: _n‘ disks of different sizes and three pegs. Initially all disks are on the first peg in order of size, the largest being on the bottom .

Problem: Move all the disks to the third peg, using the second one as an auxiliary.One can move only one disk at a time, and it is forbidden to place a larger disk on the top of a smaller one.



Recursive solution: Three steps involved are

1) First move recursively n - 1 disks from peg 1 to peg 2 (with peg 3 as auxiliary)
2) Move the largest disk directly from peg 1 to peg 3
3) Move recursively n - 1 disks from peg 2 and peg 3 (using peg 1 as auxiliary)

ALGORITHM Hanoi(n, A, C, B)

```
if n = = 1
  Move the disk from A to C
else
  Hanoi(n - 1, A, B, C)
  Move the disk from A to C
  Hanoi(n - 1, B, C, A)
```

Analysis

Input Size : n

Basic Operation : Transfer of the discs

Recurrence for number of moves: $M(n) = M(n-1) + 1 + M(n-1)$ for $n > 1$

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

**Initial Condition : ** $M(1) = 1$

Solving by backward substitution, we get

$M(n) = 2\,M(n-1) + 1$

$= 2\,(2\,M(n-2) + 1) + 1 = 2^2\,M(n-2) + 2^1 + 2^0$

$= 2^2\,(2\,M(n-3) + 1) + 2^1 + 2^0$

$= 2^3\,M(n-3) + 2^2 + 2^1 + 2^0$

…

$= 2^i\,M(n-i) + 2^{(i-1)} + \ldots + 2^1 + 2^0$

$= 2^i\,M(n-i) + 2^i - 1$

Substituting for $i = n-1$, we get

$= 2^{n-1}\,M(n-(n-1)) + 2^{n-1} - 1 = 2^{n-1}\,M(n-n+1) + 2^{n-1} - 1$

$= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1$

$= (2 \cdot 2^{n-1}) - 1 = (2^1 \cdot 2^n \cdot 2^{-1}) - 1$

$= 2^n - 1$

Example 3 : Consider the problem of finding the number of binary digits in the binary representation of a positive decimal number.

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

The Recurrence relation : $A(n) = A(\lfloor n/2 \rfloor) + 1$ for $n > 1$

Initial Condition : $A(1) = 0$

Let us consider $n = 2^k$. Then we have

Recurrence relation : $A(2^k) = A(2^{k-1}) + 1$ for $k > 0$

Initial condition : $A(2^0) = 0$

Solving by backward substitution, we have

$A(2^k) = A(2^{k-1}) + 1$; substitute $A(2^{k-1}) = A(2^{k-2}) + 1$

$= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2$; substitute $A(2^{k-2}) = A(2^{k-3}) + 1$

$= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3$; …

…

$= A(2^{k-i}) + i$

Substituting for i=k, we get

$= A(2^{k-k}) + k$

$= A(2^0) + k = A(1) + k = 0 + k$

$= k$

Since $n = 2^k$, we have $k = \log_2 n$,

$A(n) = \log_2 n \in \Theta(\log n)$.

Randomized algorithm

Randomized algorithms use random numbers or choices to decide their next step. We use these algorithms to reduce space and time complexity. There are two types of randomized algorithms:

Las Vegas algorithms

Monte-Carlo algorithms

Las Vegas Algorithms

Definition 1. A randomized algorithm is called a Las Vegas algorithm if it always returns the correct answer, but its runtime bounds hold only in expectation.

In Las Vegas algorithms, runtime is at the mercy of randomness, but the algorithm always succeeds in giving a correct answer.

Monte Carlo Algorithms

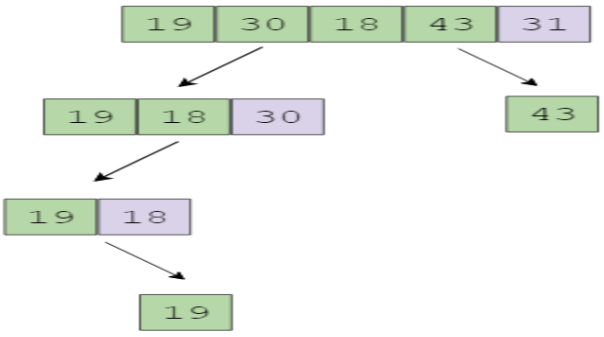
Definition 2. A randomized algorithm is called a Monte Carlo algorithm if it may fail or return incorrect answers, but has runtime independent of the randomness.

Las Vegas algorithms

Las Vegas algorithms always return correct results or fail to give one; however, its runtime may vary. An upper bound can be defined for its runtime.

Quicksort

Quicksort is a sorting algorithm that uses the divide and conquers technique. A pivot point is selected randomly from an array which is then partitioned so that all elements lesser than the pivot are placed on its left, and more significant elements are placed on its right. These steps are carried out recursively until the whole array is sorted.



The last index is selected as the pivot point in the example above. In the worst case, the pivot point selected is the maximum or minimum element.

```
def partition(array, start, end):
  pivot = array[end]
  i = start
  for j in range (start, end):
    if array[j] < pivot:
      array[i], array[j] = array[j], array[i]
```

```
        i = i+1
    array[i], array[end] = array[end], array[i]
    return i;

def QuickSort (array, start, end):
    if start < end:
      pivotelement = partition(array, start, end)
      QuickSort(array, start, pivotelement - 1)
      QuickSort(array, pivotelement + 1, end)

array=[23,11,43,22,34,42,1]
#print(len(array))
QuickSort(array, 0, len(array)-1)
print(array)
```

Monte-Carlo algorithms

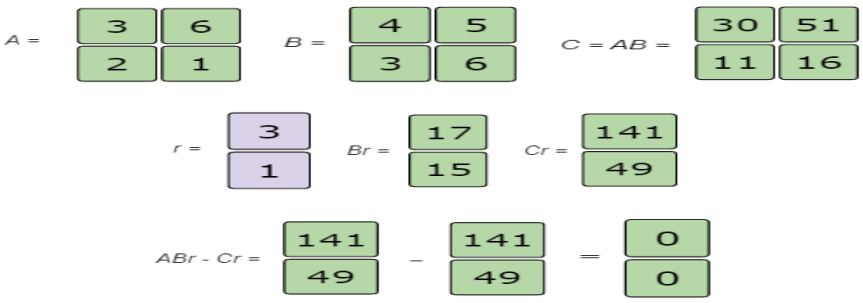
The Monte-Carlo algorithms work in a fixed running time; however, it does not guarantee correct results. One way to control its runtime is by limiting the number of iterations.

Freivalds' algorithm

Freivalds' algorithm is used to verify the result of matrix multiplication in $O(n^2)$ time. In this algorithm, three *nxn* matrices, `A`, `B`, and `C`, are given input, where `C = AB`. The algorithm generates a random matrix `r` of size n and computes $AxBr - Cr$ to verify the value of `C`. If the result is equal to zero, it means that `C` is correct, and the result is considered incorrect for non-zero values.

Through Freivalds' algorithm, we can control the runtime; however, it may produce erroneous results. For example, the result can be zero even though `C` is sometimes incorrect.

Example




```
import numpy as np
```

```
import random

def Freivald(A, B, C):
  r = ([random.randint(0,50)],[random.randint(0,50)])
  Br = np.dot(B,r)
  Cr = np.dot(C,r)
  answer = np.dot(A, Br) - Cr
  return answer

A = ([3,6],[2,1])
B = ([4,5],[3,6])
C = np.dot(A,B)
result = Freivald(A,B,C);
print (result)
```