1. What is an ALGORITHM

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

In addition, all algorithms must satisfy the following criteria:

1. Input. Zero or more quantities are externally supplied.

2. Output. At least one quantity is produced.

3. Definiteness. Each instruction is clear and produced.

4. Finiteness. If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

5. Effectiveness. Every instruction must be very basic so that it can be carried out, in principal, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.

Fig : Notion of an algorithm.

The same algorithm can be represented in several ways. Several algorithms can be used to solve the same problem. • Different ideas have different speed.

Example: Problem:GCD of Two numbers m,n

Euclids algorithm

// Based on repeatedly applying the equality

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

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

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

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

Algorithm Euclid(m,n)

//input:2 nonnegative, not-both-zero integers

//output:GCD of m and n

While n ≠ 0 do

r m mod n

m n

n r

return m

Another algorithm to solve the same problem Design and Analysis of Algorithms DSCE-CSE(Data Science)

3

Consecutive integer checking algorithm gcd(m,n)

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

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

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

stop,otherwise proceed to step4

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

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 found in Step 1&2

Step 4: Compute the product of all common factors and return as the gcd

FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

A sequence of steps involved in designing and analysing an algorithm is shown in the figure below.

(1) Understanding the Problem

This is the first step in designing of algorithm.

Read the problem's description carefully to understand the problem statement completely.

Ask questions for clarifying the doubts about the problem.

Identify the problem types and use existing algorithm to find solution.

Input (instance) to the problem and range of the input get fixed.

(2) Ascertaining the Capabilities of the Computational Device

In random-access machine (RAM), instructions are executed one after another. Accordingly, algorithms designed to be executed on such machines are called sequential algorithms.

In some newer computers, operations are executed concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called parallel algorithms.

Choice of computational devices like Processor and memory is mainly based on space and time efficiency Design and Analysis of Algorithms DSCE-CSE(Data Science)

(3) Choosing between Exact and Approximate Problem Solving

The next principal decision is to choose between solving the problem exactly or solving it approximately.

An algorithm used to solve the problem exactly and produce correct result is called an exact algorithm.

If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an approximation algorithm. i.e., produces an approximate answer. E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

(4) 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.

Algorithms+ Data Structures = Programs

Though Algorithms and Data Structures are independent, but they are combined together to develop program. Hence the choice of proper data structure is required before designing the algorithm.

Implementation of algorithm is possible only with the help of Algorithms and Data Structures

Algorithmic strategy / technique / paradigm are a general approach by which many problems can be solved algorithmically. E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and so on.

(5) Methods of Specifying an Algorithm

There are three ways to specify an algorithm. They are:

a. Natural language

b. Pseudocode

c. Flowchart

FIGURE 1.3 Algorithm Specifications

Pseudocode and flowchart are the two options that are most widely used nowadays for specifying algorithms.

a. Natural Language

It is very simple and easy to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear and thereby we get brief specification.

Example: An algorithm to perform addition of two numbers.

Step 1: Read the first number, say a.

Step 2: Read the first number, say b.

Step 3: Add the above two numbers and store the result in c.

Step 4: Display the result from c.

Such a specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of Pseudocode.

b. **Pseudocode**

Pseudocode is a mixture of a natural language and programming language constructs.

Pseudocode is usually more precise than natural language.

For Assignment operation left arrow “←”, for comments two slashes “//”,if condition, for, while loops are used.

ALGORITHM Sum(a,b)

//Problem Description: This algorithm performs addition of two numbers

//Input: Two integers a and b

//Output: Addition of two integers

c←a+b

return c

This specification is more useful for implementation of any language.

c. Flowchart Design and Analysis of Algorithms DSCE-CSE(Data Science)

5

In the earlier days of computing, the dominant method for specifying algorithms was a flowchart, this representation technique has proved to be inconvenient. Flowchart is a graphical representation of an algorithm. It is a a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

FIGURE 1.4 Flowchart symbols and Example for two integer addition.

(6) Proving an Algorithm's Correctness

Once an algorithm has been specified then its correctness must be proved.

An algorithm must 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 mod n).

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.

(7) Analyzing an Algorithm

For an algorithm the most important is efficiency. In fact, there are two kinds of algorithm efficiency. They are:

Time efficiency, indicating how fast the algorithm runs, and

Space efficiency, indicating how much extra memory it uses.

factors to analyze an algorithm are:

Time efficiency of an algorithm

Space efficiency of an algorithm

Simplicity of an algorithm

Generality of an algorithm

(8) Coding an Algorithm

The coding / implementation of an algorithm is done by a suitable programming language like C, C++, JAVA.

The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The Algorithm power should not reduced by inefficient implementation.

Standard tricks like computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.

FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

The efficiency of an algorithm can be in terms of time and space. The algorithm efficiency can be analysed by the following ways.

a. The Analysis Framework.

b. Asymptotic Notations and its properties.

c. Mathematical analysis for Recursive algorithms.

d. Mathematical analysis for Non-recursive algorithms.

a. Analysis Framework

There are two kinds of efficiencies to analyze the efficiency of any algorithm. They are:

Time efficiency, indicating how fast the algorithm runs, and

Space efficiency, indicating how much extra memory it uses.

The algorithm analysis framework consists of the following:

Units for Measuring Running Time

Orders of Growth

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

(i) Measuring an Input's Size

An algorithm's efficiency is defined 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.

For the problem of evaluating a polynomial of degree n, the size of the parameter will be the polynomial's degree or the number of its coefficients, which is larger by 1 than its degree.

In computing the product of two n × n matrices, the choice of a parameter indicating an input size does matter.

In measuring input size for algorithms solving problems such as checking primality of a positive integer n. the input is just one number and it is this number's magnitude that determines the input size. In such situations, it is preferable to measure size by the number b of bits in the n's binary representation $b=(\log_2 n)+1$.

(ii) Units for Measuring Running Time

Some standard unit of time measurement such as a second, or millisecond, and so on can be used to measure the running time of a program after implementing the algorithm.

Drawbacks

Dependence on the speed of a particular computer.

Dependence on the quality of a program implementing the algorithm.

The compiler used in generating the machine code.

The difficulty of clocking the actual running time of the program.

So, we need metric to measure an algorithm's efficiency that does not depend on these extraneous factors. One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is excessively difficult. The most important operation (+, -, *, /) of the algorithm, called the **basic operation**. Computing the number of times the basic operation is executed is easy**. The total running time is determined by basic operations count**.

Let Cop be the execution time of an algorithm's basic operation on a particular computer, and let C(n) be the number of

times this operation needs to be executed for this algorithm. Then the running time T (n) of a program implementing this algorithm on that computer by the formula

T (n) ≈ CopC(n)

The efficiency analysis framework ignores multiplicative constants and concentrates on the count's order of growth to within a constant multiple for large-size inputs.

(iii) Orders of Growth

For large values of n, it is the function's order of growth that counts just like the Table 1.1, which contains values of a few functions particularly important for analysis of algorithms Design and Analysis of Algorithms DSCE-CSE(Data Science)

(iv) Worst-Case, Best-Case, and Average-Case Efficiencies

Worst-case efficiency

The worst-case efficiency of an algorithm is its efficiency for the worst case input of size n.

The algorithm runs the longest among all possible inputs of that size.

For the input of size n, the running time is Cworst(n) = n.

Best case efficiency

The best-case efficiency of an algorithm is its efficiency for the best case input of size n.

The algorithm runs the fastest among all possible inputs of that size n.

In sequential search, If we search a first element in list of size n. (i.e. first element equal to a search key), then the running time is Cbest(n) = 1

Average case efficiency

The Average case efficiency lies between best case and worst case.

To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size n.

The standard assumptions are that

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

o The probability of the first match occurring in the ith position of the list is the same for every i.

In the case of a successful search, the probability of the first match occurring in the ith 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,

Asymptotic and Basic Efficiency Classes

Asymptotic Notations

Asymptotic notations are used to represent the growth of an algorithm as the input increases. The algorithm whose growth rate is less with the increase in problem size is considered as the better one.

Asymptotic notation is a way of comparing functions that ignores constant factors and small input sizes. Three notations used to compare orders of growth of an algorithm's basic operation count are:**O, Ω, Θ notations.**

Big Oh- O notation

Definition: Design and Analysis of Algorithms DSCE-CSE(Data Science)

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

Ex:

• express the upper bound of an algorithm's running time

• measures the worst case time complexity or the longest amount of time an algorithm can take to complete

Big Omega- Ω notation

Definition:

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**

Ex: -n3€ Ω **O(n2), 100n+5** € Ω(n2)

Big Theta- Θ notation

Definition:

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 constant multiple 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**

ex: ½ n(n-1)€ Θ(n2) Design and Analysis of Algorithms DSCE-CSE(Data Science)

9

FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

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

• running time (time efficiency)

• memory space (space efficiency)

Time being more critical than space, we concentrate on Time efficiency of algorithms. The theory developed, holds good for space complexity also.

Experimental Studies: requires writing a program implementing the algorithm and running the program with inputs of varying size and composition. It uses a function, like the built-in clock() function, to get an accurate measure of the actual running time, then analysis is done by plotting the results.

Basic Efficiency classes

The time efficiencies of a large number of algorithms fall into only a few classes.

Useful Property Involving the Asymptotic Notations Design and Analysis of Algorithms DSCE-CSE(Data Science)

10

Using Limits for Comparing Orders of Growth

Three principal cases may arise: Design and Analysis of Algorithms DSCE-CSE(Data Science)

11

MATHEMATICAL ANALYSIS (TIME EFFICIENCY) OF NON-RECURSIVE ALGORITHMS

General plan for analyzing efficiency of non-recursive algorithms:

1. Decide on parameter n indicating **input 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 the type of input, investigate **worst, average, and best case efficiency** separately.

4. Set up **summation** for C(n) reflecting the number of times the algorithm's basic operation is executed.

5. Simplify summation using standard formulas and establish the order of growth

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

//Determines the value of 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

Max =*A*[0]

for *i* = 1 to *n* -1 do

if *A*[*i*] > *Max*

Max = A[i]

return *Max*

Analysis:

1. Input size: number of elements = n (size of the array)

2. Basic operation:

a) Comparison

b) Assignment

3. NO best, worst, average cases.

4. Let C (*n*) denotes number of comparisons: Algorithm makes one comparison on

each execution of the loop, which is repeated for each value of the loop's variable

i within the bound between 1 and n – 1. Design and Analysis of Algorithms DSCE-CSE(Data Science)

12

Example: Element uniqueness problem

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

1. Input size: number of elements = n (size of the array)

2. Basic operation: Comparison

3. Best, worst, average cases EXISTS.

Worst case input is an array giving largest comparisons.

• Array with no equal elements

• Array with last two elements are the only pair of equal elements

4. Let C (*n*) denotes number of comparisons in worst case: Algorithm makes one

comparison for each repetition of the innermost loop i.e., for each value of the

loop's variable j between its limits *i + 1 and n – 1*; and this is repeated for each value of the outer loop i.e, for each value of the loop's variable i between its

limits *0 and n – 2* Design and Analysis of Algorithms DSCE-CSE(Data Science)

13

Example:

Algorithm Matrixmult(A[0..n-1,0..n-1], B[0..n-1,0..n-1])

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

n-1 n-1 n-1

M(n) = Σ Σ Σ 1 = n 3

I=0 j=0 k=0

MATHEMATICAL ANALYSIS (TIME EFFICIENCY) OF RECURSIVE ALGORITHMS

General plan for analysing efficiency of recursive algorithms:

1. Decide on parameter n indicating **input 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 the type of input, investigate **worst, average, and best case efficiency** separately.

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

5. **Solve** the recurrence or establish the order of growth

Example: Factorial function

ALGORITHM *Factorial (n)*

//Computes n! Recursively

//Input: A nonnegative integer n

//Output: The value of n!

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

else return Factorial (*n* – 1) * *n*

Analysis:

1. Input size: given number = n

2. Basic operation: multiplication

3. NO best, worst, average cases.

4. Let M (*n*) denotes number of multiplications.

M (*n*) = M (*n* – 1) + 1 for *n* > 0

M (0) = 0 *initial condition*

Where: M (*n* – 1) : to compute Factorial (*n* – 1)

1 :to multiply Factorial (*n* – 1) by *n*

5. Solve the recurrence: Solving using *"Backward substitution method":*

M (*n*) = M (*n* – 1) + 1

= [M (*n* – 2) + 1] + 1 Design and Analysis of Algorithms DSCE-CSE(Data Science)

= M (*n* – 2) + 2

= [M (*n* – 3) + 1] + 3

= M (*n* – 3) + 3

…

In the ith recursion, we have

= M (*n* – *i*) + *i*

When i = n, we have

= M (*n* – *n*) + *n* = M (0) + *n*

Since M (0) = 0

= n

Example: 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:

1. Input size: given number = n

2. Basic operation: addition

3. NO best, worst, average cases.

4. Let A (*n*) denotes number of additions.

A (*n*) = A (⌊n/2⌋) + 1 for *n* > 1

A (1) = 0 *initial condition*

Where: A (⌊n/2⌋): to compute *BinRec* (⌊n/2⌋)

1 : to increase the returned value by 1

5. Solve the recurrence:

A (*n*) = A (⌊n/2⌋) + 1 for *n* > 1

Example: Tower of Hanoi

We have n disks of different sizes that can slide onto any of three pegs. Consider A (source), B (auxiliary), and C (Destination). 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.

Solution to Tower of Hanoi: To move n>1 disks from peg1 to peg3, first move recursively n-1 disks from peg1 to peg2, then move largest disk from peg1 to peg3,then move recursively n-1 disks from peg2 to peg3 Design and Analysis of Algorithms DSCE-CSE(Data Science)

2. BRUTE FORCE

Introduction

Brute force is a straightforward approach to problem solving, usually directly based on the problem's statement and definitions of the concepts involved. For some important problems (e.g., sorting, searching, string matching), the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size Even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem. A brute-force algorithm can serve an important theoretical or educational purpose.

the brute-force approach should not be overlooked as an important algorithm design strategy.

- brute force is applicable to a very wide variety of problems.

- for some important problems—e.g., sorting, searching, matrix multiplication, string matching—the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size.

 the expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with acceptable speed.

 even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem.

 a brute-force algorithm can serve an important theoretical or educational purpose as a yardstick with which to judge more efficient alternatives for solving a problem.

Brute force Method - Strengths and Weakness

Strengths:

– Wide applicability

– Simplicity

– Yields reasonable algorithms for some important problems

• Searching

• String matching

• Matrix multiplication

– Yields standard algorithms for simple computational tasks

• Sum/product of n numbers

• Finding max/min in a list

Weaknesses:

– Rarely yields efficient algorithms

– Some brute force algorithms unacceptably slow

– Not as constructive/ creative as some other design techniques

Sorting Problem Brute force approach to sorting

Problem:Given a list of n orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in nondecreasing order.

Selection Sort

ALGORITHM SelectionSort(A[0..n - 1])

//The algorithm sorts a given array by selection sort

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

//Output: Array A[0..n - 1] sorted in ascending order

for i = 0 to n - 2 do

min = i

for j _=i + 1 to n - 1 do

if A[j]<A[min] min= j

swap A[i] and A[min]

Example: Design and Analysis of Algorithms DSCE-CSE(Data Science)

17

Performance Analysis of the selection sort algorithm: The input's size is given by the number of elements n.

The algorithm's basic operation is the key comparison A[j]<A[min]. The number of times it is executed depends only on the array's size and is given by

Thus, selection sort is a O(n2) algorithm on all inputs. The number of key swaps is only O(n) or, more precisely, n-1 (one for each repetition of the i loop).This property distinguishes selection sort positively from many other sorting algorithms.

Bubble Sort

Compare adjacent elements of the list and exchange them if they are out of order. Then we repeat the process, By doing it repeatedly, we end up 'bubbling up' the largest element to the last position on the list

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

//The algorithm sorts array A[0..n - 1] by bubble sort

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

//Output: Array A[0..n - 1] sorted in ascending order

for i =0 to n - 2 do

for j = 0 to n - 2 - i do

if A[j + 1]<A[j]

swap A[j] and A[j + 1

Example Design and Analysis of Algorithms DSCE-CSE(Data Science)

18

The first 2 passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm

Bubble Sort analysis

The number of key swaps depends on the input. For the worst case of decreasing arrays, it is the same as the number of key comparisons.

Observation: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm Though the new version runs faster on some inputs, it is still in O(n2) in the worst and average cases. Bubble sort is not very good for big set of input.How ever bubble sort is very simple to code.

General Lesson From Brute Force Approach

A first application of the brute-force approach often results in an algorithm that can be improved with a modest amount of effort.Compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search).

Sequential Search

ALGORITHM SequentialSearch2(A[0..n], K)

//The algorithm implements sequential search with a search key

as a // sentinel

//Input: An array A of n elements and a search key K

//Output: The position of the first element in A[0..n - 1] whose

value is Design and Analysis of Algorithms DSCE-CSE(Data Science)

19

// equal to K or -1 if no such element is found

A[n] =K

I= 0

A[i] = K do

i = i + 1

if i < n return i

else return -1

Brute-Force String Matching

Given a string of n characters called the text and a string of m characters (m = n) called the pattern, find a substring of the text that matches the pattern. To put it more precisely, we want to find i—the index of the leftmost character of the first matching substring in the text—such that

ti = p0, . . . , ti+j = pj , . . . , ti+m-1 = pm-1:

t0 . . . ti . . . ti+j . . . ti+m-1 . . . tn-1 text T

p0 . . . pj . . . pm-1 pattern P

1. Pattern: 001011

Text: 10010101101001100101111010

2. Pattern: happy

Text: It is never too late to have a happy

childhood.

The algorithm shifts the pattern almost always after a single character comparison. in the worst case, the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the n - m + 1 tries.Thus, in the worst case, the algorithm is in θ(nm). Design and Analysis of Algorithms DSCE-CSE(Data Science)

Module-2

Divide and Conquer

INTRODUCTION

Divide-and-conquer algorithms work according to the following general plan:

1. Divide: A problem is divided into several subproblems of the same type, ideally of about equal size.

2. Conquer: The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).

3. Combine: If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

The divide-and-conquer technique is shown in Figure

In the most typical case of divide-and-conquer a problem's instance of size n is divided into two instances of size n/2. More generally, an instance of size n can be divided into 'b' instances of size n/b, with 'a' of them needing to be solved.

T (n) = aT (n/b) + f (n)

where f (n) is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions. Recurrence relation is called the general divide-and-conquer recurrence.

Some of the algorithms that make use of divide and conquer approach are Binary search, Quick Sort, Merge Sort, Defective chess board problem etc., Design and Analysis of Algorithms DSCE-CSE(Data Science)

Mergesort : Mergesort sorts a given array A[0…n − 1] by dividing it into two halves A[0… ⌊n/2⌋ − 1] and A[⌊n/2⌋ …n − 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

The merging of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to

are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array. Design and Analysis of Algorithms DSCE-CSE(Data Science)

Analysis of Merge Sort

• Running time T(n) of Merge Sort:

• Divide: computing the middle takes Θ(1)

• Conquer: solving 2 subproblems takes 2T(n/2)

• Combine: merging n elements takes Θ(n)

• Total:

$T(n) = \Theta(1)$ if $n = 1$

$T(n) = 2T(n/2) + \Theta(n)$ if $n > 1$

Using Master Theorem , a=2,b=2 d=1, Hence, $T(n) = \Theta(n \log n)$

Quick Sort:

Quicksort is based on the divide-and conquer approach.

 A partition is used to divide the problem that is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

 The strategy used for selecting the pivot element is : the subarray's first element: p = A[l]

 Scan the subarray from both ends, comparing the subarray's elements to the pivot.

 The left-to-right scan, denoted by index i, starts with the second element. This scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot.

 The right-to-left scan, denoted by index j, starts with the last element of the subarray. This scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

Design and Analysis of Algorithms DSCE-CSE(Data Science)

23

Fig (a) shows the working of Quicksort. Fig(b) shows the Tree of recursive calls to Quicksort with input values l and r of subarray bounds and split position s of a partition obtained.
Design and Analysis of Algorithms DSCE-CSE(Data Science)

24

Pseudo code

Best case input: If all the partitions happen in the middle of corresponding subarrays

Worst case input: For increasing arrays, i.e for inputs that are already solved.

Best case efficiency:

Number of key comparisons is n + 1 if the scanning indices cross over

and is n if they coincide

The number of key comparisons in the best case satisfies the recurrence

Worst case efficiency:

In the worst case, all the partitions will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned.

This will happen, in particular, for increasing arrays, i.e., for inputs for which the problem is already solved

If we use A[0] as the pivot, the left-to-right scan will stop on A[1] while the right-to-left scan will go all the way to reach A[0], indicating the split at position 0: Design and Analysis of Algorithms DSCE-CSE(Data Science)

25

Average case efficiency:

Let Cavg(n) be the average number of key comparisons made by quicksort on a randomly ordered array of size n. A partition can happen in any position s ($0 \le s \le n-1$) after n+1comparisons are made to achieve the partition.

After the partition, the left and right subarrays will have s and $n - 1 - s$ elements, respectively. Assuming that the partition split can happen in each position s with the same probability 1/n, we get the following recurrence relation

Strassen's Matrix Multiplication

Strassen in 1969 which gives an overview that how we can find the multiplication of two 2*2 dimension matrix by the brute-force algorithm. But by using divide and conquer technique the overall complexity for multiplication two matrices is reduced. This happens by decreasing the total number if multiplication performed at the expenses of a slight increase in the number of addition.

This is accomplished by using the following formulas:

Thus, to multiply two 2 × 2 matrices, Strassen's algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions.

Let A and B be two n × n matrices where n is a power of 2.We can divide A,B, and their product C into four n/2 × n/2 submatrices each as follows: Design and Analysis of Algorithms DSCE-CSE(Data Science)