Analysis & Design of Algorithms

Course Code	BCS401	Semester	4
Teaching Hours/Week (L: T:P: S)	3:0:0:0	CIE Marks	50
Total Hours of Pedagogy	40	SEE Marks	50
Credits	03	Total Marks	100
Examination type (SEE)		Exam Hours	03
		Theory	

Course objectives:

- To learn the methods for analyzing algorithms and evaluating their performance.
- To demonstrate the efficiency of algorithms using asymptotic notations.
- To solve problems using various algorithm design methods, including brute force, greedy, divide and conquer, decrease and conquer, transform and conquer, dynamic programming, backtracking, and branch and bound.
- To learn the concepts of P and NP complexity classes.

Teaching-Learning Process (General Instructions)

These are sample Strategies, which teachers can use to accelerate the attainment of the various course outcomes.

1. Lecturer method (L) does not mean only the traditional lecture method, but different types of teaching methods may be adopted to achieve the outcomes.
2. Utilize video/animation films to illustrate the functioning of various concepts.
3. Promote collaborative learning (Group Learning) in the class.
4. Pose at least three HOT (Higher Order Thinking) questions in the class to stimulate critical thinking.
5. Incorporate Problem-Based Learning (PBL) to foster students' analytical skills and develop their ability to evaluate, generalize, and analyze information rather than merely recalling it.
6. Introduce topics through multiple representations.
7. Demonstrate various ways to solve the same problem and encourage students to devise their own creative solutions.
8. Discuss the real-world applications of every concept to enhance students' comprehension.

Module-1

INTRODUCTION: What is an Algorithm?, Fundamentals of Algorithmic Problem Solving.
FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY: Analysis Framework, Asymptotic Notations and Basic Efficiency Classes, Mathematical Analysis of Non recursive Algorithms, Mathematical Analysis of Recursive Algorithms.
BRUTE FORCE APPROACHES: Selection Sort and Bubble Sort, Sequential Search and Brute Force String Matching.
Chapter 1 (Sections 1.1,1.2), Chapter 2(Sections 2.1,2.2,2.3,2.4), Chapter 3(Section 3.1,3.2)

Module-1:Introduction

LECTURE 1:

1.1 Introduction

What is an Algorithm?

Algorithm: An algorithm is a finite sequence of un ambiguous instructions to solve a particular problem

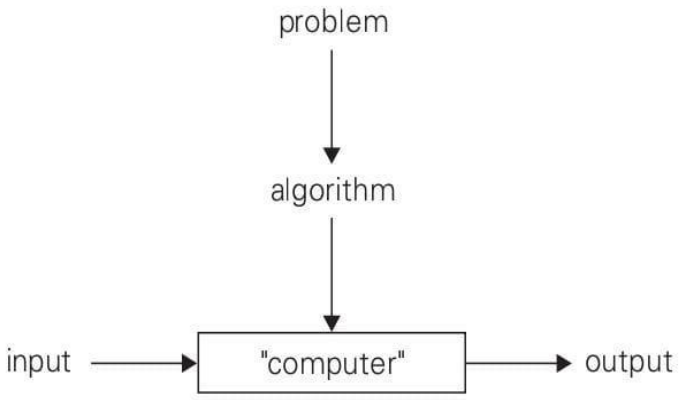


FIGURE 1.1 The notion of the algorithm.

Characteristics of algorithm

- **Input**: Zero or more quantities are externally supplied.
- **Output**: Atleast one quantity is produced.
- **Definiteness**: Each instruction is clear and unambiguous. It must be perfectly clear what should be done.
- **Finiteness**: If we trace out the instruction of an algorithm ,then for all cases, the algorithm terminates after a finite number of steps.

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

Algorithm Specification

An algorithm can be specified in

1) Simple English
2) Graphical representation like flowchart
3) Programming language like c++/java
4) Combination of above methods.

Steps for writing an algorithm:

1. An algorithm is a procedure. It has two parts; the first part is **head** and the second part is **body**.
2. The Head section consists of keyword **Algorithm** and Name of the algorithm with Parameter list E.g. Algorithm name1(p1,p2,…,p3)

The head section also has the following:

//Problem Description:

//Input:

//Output:

3. In the body of algorithm various programming constructs like **if, for ,while** and some statements like assignments are used.
4. The compound statements may been closed with**{**and**}** brackets.**if**,**for**,**while** can be closed by **end if**, **end for**, **end while** respectively. Proper indention is must for block.
5. Comments are written using//at the beginning.
6. The**identifier**shouldbeginbyaletterandnotbydigit.Itcontainsalphanumericletters after first letter. No need to mention data types.
7. Theleftarrow**"←"**usedasassignmentoperator.E.g.v←10
8. **Boolean**operators(TRUE,FALSE),**Logical**operators(AND,OR,NOT)and**Relational** operators(<,<=,>,>=,=,≠,<>)arealsoused.
9. Inputand Outputcanbe doneusing **read**and**write**.
10. **Array[]**,**ifthen elsecondition**,**branch**and**loop**canbealsousedinalgorithm.

Example:

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

Euclid's algorithm is based on applying repeatedly the equality 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.Sincegcd*(m,* 0*)* = *m*, the last value of *m* is also the greatest common divisor of the initial *m* and *n*.

gcd*(*60,24*)*can be computed as follows: gcd*(*60,24*)*=gcd*(*24,12*)*=gcd*(*12,0*)*=12.

Euclid's algorithm for computing gcd*(m,n)*

Step 1: If*n*=0,return the value of *m* as the answer and stop; otherwise, proceed to Step 2.

Step2: Divide *m* by *n* and assign the value of the remainder o*r*.

Step3: Assign the value of *n* to *m* and the value of *r* to *n*. Goto Step1. Alternatively, we can express the same algorithm in pseudocode:

Euclid's algorithm for computing gcd*(m, n)* expressed in pseudocode

ALGORITHM *Euclid_gcd(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
```

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

Step1 : Assign the value of min{*m,n*}to *t.*

Step2: Divide *m by t.*If the remainder of this division is 0, go to Step 3; otherwise, go

to Step 4.

Step3: 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 Step4.

Step4: Decrease the value of *t* by 1.Goto Step 2.

Note that unlike Euclid's algorithm, this algorithm, in the form presented, does not work correctly when one of its input numbers is zero. This example illustrateswhyitissoimportanttospecifythesetofanalgorithm'sinputsexplicitly and carefully.

1.2FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

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

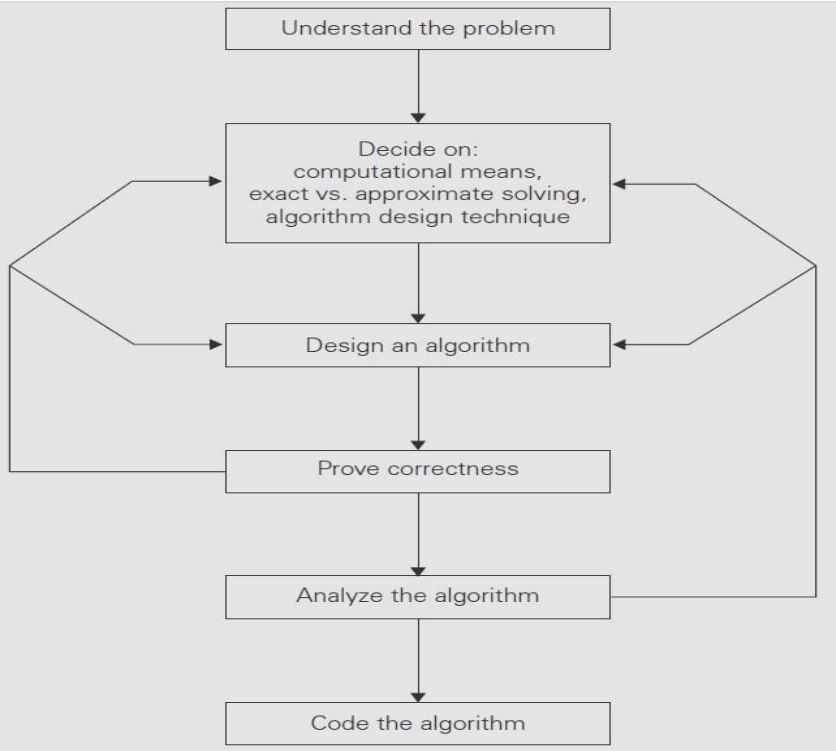


FIGURE1.2 Algorithmdesignandanalysis process.

(i) 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.

(ii) Decision making

The Decision making is done on the following:

(a) As certaining the Capabilities of the Computational Device

- In ***random-access machine*** (***RAM***), instructions areexecuted one after another (The central assumption is that one operation at a time). Accordingly, algorithmsdesigned 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***.

- ChoiceofcomputationaldeviceslikeProcessorandmemoryismainlybasedon **Space and time efficiency**

(b) Choosing between Exact and Approximate Problem Solving

- Then ext 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.

(c) 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*+DataStructures=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.

(iii)Methods of Specifying an Algorithm

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

a. **Natural language**
b. **Pseudocode**
c. **Flowchart**

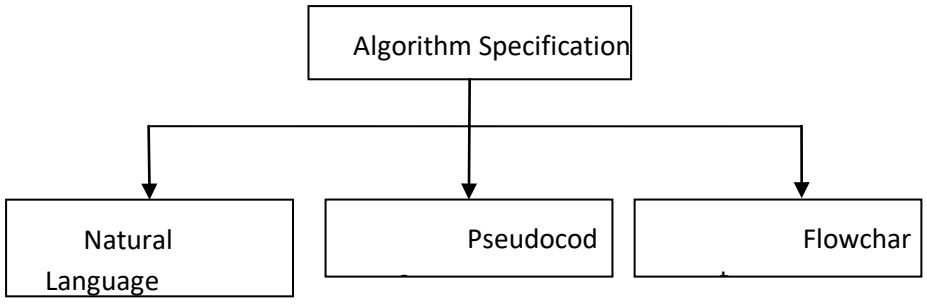


FIGURE1.3 Algorithm Specifications

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.

> Step1:Read the first number, say a.
> Step2: Read the first number, say b.
> Step3: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
```

This specification is more useful for implementation of any language.

c. Flowchart

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. Itis a a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

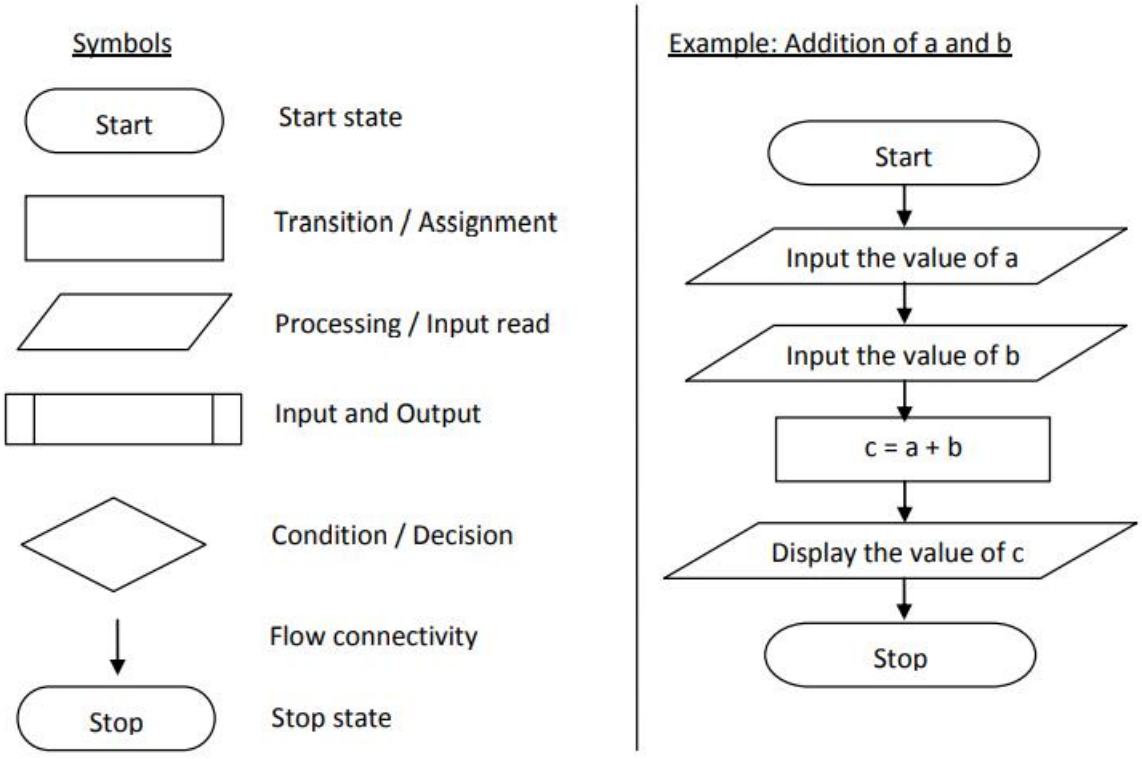


FIGURE 1.4 Flowchart symbols and Example for two integer addition.

(iv) Proving an Algorithm's Correctness

- Once an algorithm has been specified t 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 Divisors terms 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.
- The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The **error** produced by the algorithm should not exceed a pre defined limit.

(v) 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.
- The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.
- Sof factors to analyze an algorithm are:
    - Time efficiency of an algorithm
    - Space efficiency of an algorithm
    - Simplicity of an algorithm
    - Generality of an algorithm

(vi) Coding an Algorithm

- The coding / implementation of an algorithm is done by a suitable programming languagelike 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.
- Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by **orders of magnitude**. But once an algorithm is selected, a 10–50% speedup may be worth an effort.
- It is very essential to write an **optimized code (efficient code)** to reduce the burden of compiler.

Review Questions

1.What is an algorithm?

2.What are the 3 methods of specifying an algorithm

3.List the steps involved in fundamentals of problem solving?

LECTURE 2:

1.3 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 analyzed by the following ways.

a. Analys is Framework.
b. A symptotic Notations and its properties.
c. Mathematical analysis for Recursive algorithms.
d. Mathematical analysis for Non- recursive algorithms.

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 analys is frame work consists of the following:

- Measuring an Input's Size
- 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. In most cases, selecting such a parameter is quite straight forward. For example, it will be the size of the list for problems of sorting, searching.
- For the problem of evaluating a polynomial $p(x) = a_n x^n + \ldots + a_0$ 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 $n \times n$ matrices, the choice of a parameter indicating an input size does matter.
- Consider a spell-checking algorithm. If the algorithm examines individual characters of its input, then the size is measured by the number of characters.
- In measuring input size for algorithms solving problems such as checking primality of a positive integer n. the input is just one number.
- The input size by the number b of bits in the n's binary representation is $b=(\log_2 n)+1$.

(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.

Orders of Growth

- A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.
- For example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid's algorithm is compared to the other algorithms, the difference in algorithm efficiencies becomes clear for larger numbers only.
- For large values of *n*, it is the function's order of growth that counts just like the Table1.1,

Which contains values of a few functions particularly important for analysis of algorithms.

TABLE1.1 Values(approximate)of several functions important for analysis of algorithms

n	$\sqrt{n}$	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
1	**1**	**0**	**1**	**0**	**1**	**1**	**2**	**1**
2	**1.4**	**1**	**2**	**2**	**4**	**4**	**4**	**2**
4	**2**	**2**	**4**	**8**	**16**	**64**	**16**	**24**
8	**2.8**	**3**	**8**	**$2.4 \cdot 10^1$**	**64**	**$5.1 \cdot 10^2$**	**$2.6 \cdot 10^2$**	**$4.0 \cdot 10^4$**
10	**3.2**	**3.3**	**10**	**$3.3 \cdot 10^1$**	**10^2**	**10^3**	**10^3**	**$3.6 \cdot 10^6$**
16	**4**	**4**	**16**	**$6.4 \cdot 10^1$**	**$2.6 \cdot 10^2$**	**$4.1 \cdot 10^3$**	**$6.5 \cdot 10^4$**	**$2.1 \cdot 10^{13}$**
10^2	**10**	**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	31	10	10^3	$1.0 \cdot 10^4$	10^6	10^9	Very big computation	
10^4	10^2	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	$3.2 \cdot 10^2$	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	10^3	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

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

Consider Sequential Search algorithm some search key *K*

ALGORITHM *SequentialSearch(A[0..n - 1], K)*

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

Clearly, the running time of this algorithm can be quite different for the same list size *n*.

In the worst case, there is no matching of elements or the first matching element can found at last on the list. In the best case, there is matching of elements at first on the list.

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

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 toa search key),then the running time is $C_{best}(n) = 1$

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

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1-p)$$
$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p)$$
$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p).$$

Yet another type of efficiency is called ***a mortised efficiency***. It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure.

1.4ASYMPTOTIC NOTATIONS AND ITS PROPERTIES

A symptotic notation is a notation, which is used to take meaningful statement about the efficiency of a program.

The efficiency analysis frame work concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency.

To compare and rank such orders of growth, computer scientists use three notations, they are:

- O-Big oh notation
- Ω-Big omega notation
- Θ-Big theta notation

Let *t(n)*and *g(n)*can be any non negative functions defined on these to f natural numbers. The algorithm's running time *t(n)*usually indicated by its basic operation count *C(n)*,and *g(n)*, Some simple function to compare with the count.

Example1:

$$n \in O(n^2), \qquad 100n + 5 \in O(n^2), \qquad \frac{1}{2}n(n-1) \in O(n^2).$$

Where $g(n)=n^2$

(i) O-Big oh notation

A function t*(n)* is said to be in *O(g(n))*, denoted $t\ (n)\in O(g(n))$, if *t (n)* is bounded above by some constant multiple of *g(n)* for all large *n*, i.e., if there exist some positive constant *c* and some nonnegative integer n_0 such that

$$(n)\leq cg(n)foralln\geq n_0.$$

Where*t(n)*and*g(n)*arenonnegativefunctionsdefinedonthesetofnaturalnumbers. O = Asymptotic upper bound = Useful for worst case analysis = Loose bound

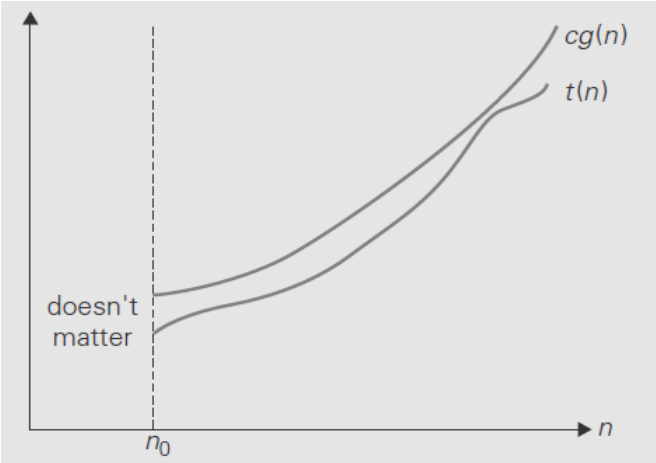


FIGURE1.5 Big-ohnotation: $(n)\in O(g(n))$.

Example2: Provetheassertions$100n+5\in(n^2)$.

Proof: $100n+5\leq100n+ n$ (forall $n\geq5$)

$=101n$

$\leq101n^2$(Ən$\leq n^2$)

Since, the definition gives us a lot of freedom in choosing specific values for constants **c** And $\mathbf{n_0}$. We have c=101 and $n_0=5$

Example3: Prove the assertions $100\ n+5\in(n)$.

Proof: $100n+5\leq100n+ 5n$ (forall$n\geq1$)

$=105n$

i.e., $100n+5\leq105n$

i.e., $t(n)\leq cg(n)$

ë$100n+5\in(n)$with c=105and$n_0=1$

(ii) Ω-Big omega 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 positive constant multiple of *g(n)* for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$t(n) \geq cg(n)$ for all $n \geq n_0$.

Where *t(n)* and *g(n)* are nonnegative functions defined on the set of natural numbers.

Ω=A symptotic lower bound=Useful for best case analysis= Loose bound

Ω=A symptotic lower bound=Useful for best case analysis= Loose bound

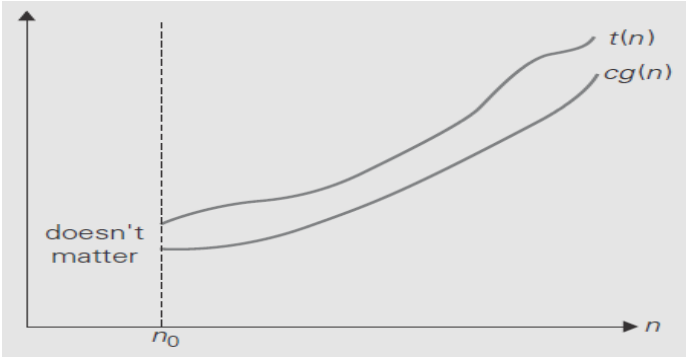


FIGURE1.6 Big-omega notation: $t(n) \in \Omega(g(n))$.

Example4: Prove the assertions $n^3+10n^2+4n+2 \in \Omega(n^2)$.

Proof: $n^3+10n^2+4n+2 \geq n^2$ (for all $n \geq 0$)

i.e., by definition $(n) \geq cg(n)$, where $c=1$ and $n_0=0$

(iii) Θ-Big theta notation

A function (n) is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, If t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

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

Where *t(n)* and *g(n)* are non negative functions defined on these to f natural numbers.

Θ=A symptotic tight bound =Useful for average case analysis

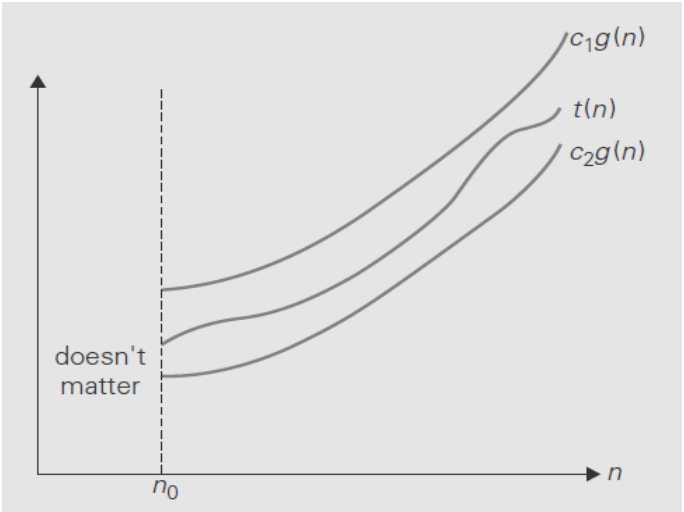


FIGURE1.7 Big- theta notation : $t(n) \in \Theta(g(n))$.

Example 5: Prove the assertions $\frac{1}{2}n(n-1) \in \Theta(n^2)$.

Proof: First prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \quad \text{for all } n \geq 0.$$

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

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \left[\frac{1}{2}n\right]\left[\frac{1}{2}n\right] \quad \text{for all } n \geq 2.$$

$$\therefore \quad \frac{1}{2}n(n-1) \geq \frac{1}{4}n^2$$

$$\text{i.e.,} \quad \frac{1}{4}n^2 \leq \frac{1}{2}n(n-1) \leq \frac{1}{2}n^2$$

Hence, $\frac{1}{2}n(n-1) \in \Theta(n^2)$, where $c2=\frac{1}{4}$, $c1=\frac{1}{2}$ and $n_0=2$

Note: asymptotic notation can be thought of as "relational operators" for functions similar to the corresponding relational operators for values.

$= \Rightarrow \Theta()$, $\leq \Rightarrow O()$, $\geq \Rightarrow \Omega()$, $< \Rightarrow o()$, $> \Rightarrow \omega()$

Useful Property Involving the Asymptotic Notations

The following property, in particular, is useful in analyzing algorithms that comprise two consecutively executed parts.

THEOREM: If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n)+t_2(n) \in O(\max\{g_1(n), g_2(n)\})$. (The analogous assertions are true for the Ω and Θ notations as well.)

PROOF: 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 $a_1 \leq b_1$ and $a_2 \leq b_2$, then $a_1 + a_2 \leq 2 \max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some non negative integer n_1 such that

$$t_1(n) \leq c_1 g_1(n) \text{ for all } n \geq n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \leq c_2 g_2(n) \text{ for all } n \geq n_2.$$

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

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

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

The property implies that the algorithm's overall efficiency will be determined by the part With a higher order of growth, i.e. ,its least efficient part.

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

Review Questions:

1.what does Time and space efficiency indicate?

2.Write A symptotic notations and its properties

3.What is the running time in worst case, best case, and average case?

LECTURE 3:

1.5 MATHEMATICAL ANALYSIS FOR NON-RECURSIVE ALGORITHMS

General Plan for Analyzing the Time Efficiency of Non recursive Algorithms

1. Decide on a *parameter*(or parameters)indicating an input's size.
2. Identify the algorithm's *basic operation*(in the inner most loop).
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 to be investigated separately.
4. Setup a *sum* expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation either find a closed form formula for the count or at the least, establish its *order of growth*.

EXAMPLE 1: Consider the problem of finding the value of **the largest element in a list of n numbers**. Assume that the list is implemented as an array for simplicity.

ALGORITHM Max Element(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
Max val ←A[0]
For i ←1 **to** n −1 **do**
If A[i]> max val
Max val← A[i]
Return max val

Algorithm analysis

- The measure of an input's size here Is the number of elements in the array, i.e., n.
- There are two operations in the for loop's body:
  - The comparison A[i]>max val and
  - The assignment max val←A[i].

- The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.
- The number of comparisons will be the same for all arrays of size n; therefore, there is no need to distinguish among the worst, average, and best cases here.
- Let C(n) denotes the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n − 1, inclusive. Therefore, the sum for C(n) is calculated as follows:

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

i.e., Sum up 1 in repeated n-1 times

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

EXAMPLE2: Consider the **element uniqueness problem**: check whether all the Elements in a given array of n elements are distinct.

ALGORITHM Unique Elements(A[0..n−1])

//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n−1]
//Output: Returns "true" I fall 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

Algorithm analysis

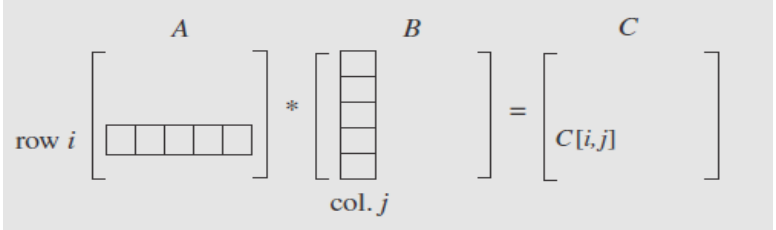
- The natural measure of the input's size here is a gain n (the number of elements in the array).
- Sincetheinnermostloopcontainsasingleoperation(thecomparisonoftwoelements),we

Should consider it as the algorithm's basic operation.

- The number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only**.**
- 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.

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$
$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

EXAMPLE3: Consider matrix multiplication. Given two n× n matrices A and B, find the time efficiency of the definition-based algorithm for computing their product C=AB. By definition, C is an n×n matrix whose elements are computed as the scalar(dot)products of the rows of matrix A and the columns of matrix B:



whereC[i,j]=A[i,0]B[0,j]+...+A[i,k]B[k,j]+...+A[i,n−1]B[n−1,j]forevery pairof indices $0 \le i, j \le n-1$.

ALGORITHM Matrix Multiplication(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

fork←0 to n − 1 do

C[i,j]←C[i,j]+A[i, k]∗B[k, j]

Return C

Algorithm analysis

- An input's size is matrix order n.
- There are two arithmetical operations (multiplication and addition) in the innermost loop. But we consider multiplication as the basic operation.
- Let us set up a sum for the total number of multiplications M(n) executed by the algorithm. Since this count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.
- There is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable k ranging from the lower bound 0 to the upper bound n − 1.
- Therefore, the number of multiplications made for every pair of specific values of variables i and j is

$$\sum_{k=0}^{n-1} 1$$

ThetotalnumberofmultiplicationsM(n)isexpressedbythefollowingtriplesum:

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

Now, we can compute this sum by using formula(S1) and rule (R1)

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3$$

.

The running time of the algorithm on a particular machine m, we can do it by the product

$$T(n) \approx c_m M(n) = c_m n^3,$$

If we consider, time spent on the additions too, then the total time on the machine is

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3$$

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

Algorithm analysis

- An input's size is n.
- The loop variable takes on only a few values between its lower and upper limits.
- Since the value of n is about halved one achrepetition of the loop, the answer should be about $\log_2 n$.
- The exact formula for the number of times.
- The comparison $n>1$willbeexecutedisactually$\lfloor \log_2 n \rfloor +1$.

Review Questions

1.Consider the element uniqueness problem. Check whether all the elements in a given array of n elements are distinct

2.Consider matrix multiplication . Given t won× n matrices A and B, find the time efficiency of the definition-based algorithm for computing their product C=AB.B y definition ,C

is an n×n matrix whose elements are computed as the scalar (dot)products of the rows of matrix A and the columns of matrix B:

3.write an algorithm for finding the value of the largest element in a list of n numbers. Assume that the list is implemented as an array for simplicity

LECTURE 4:

1.5MATHEMATICAL ANALYSIS FOR RECURSIVE ALGORITHMS

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

6. Decide on a *parameter*(or parameters)indicating an input's size.
7. Identifythealgorithm's*basic operation*.
8. 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.
9. *Set up a recurrence relation*, with an appropriate initial condition, for the number of times the basic operation is executed.
10. Solve the recurrence or, atleast, ascertain the *order of growth* of its solution.

EXAMPLE1:Compute the factorial function F(n)=n!for an arbitrary nonnegative integer n. Since $n!=1\cdot\ldots\cdot(n-1)\cdot n=(n-1)!\cdot n$,forn≥1and0!=1by definition,

we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm.

ALGORITHM *F(n)*

//Computes *n*! recursively

//Input: A nonnegative integer *n*

//Output: The value of *n*!

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

else return $F(n-1)*n$

Algorithm analysis

- For simplicity, we consider itself as an indicator of this algorithm's input size.i.e.1.
- The basic operation of the algorithm is multiplication, whose number of executions we denote $M(n)$.Since the function $F(n)$ is computed according to the formula $F(n)=F(n-1)\bullet n$ for $n>0$.
- The number of multiplications $M(n)$ needed to compute it must satisfy the equality

$$M(n) = \underbrace{M(n-1)}_{\text{To compute } F(n-1)} + \underbrace{1}_{\text{To multiply } F(n-1) \text{ by } n} \quad \text{for } n > 0$$

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

Recurrence relations

The last equation defines the sequence M(n) that we need to find. This equation defines M(n) not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely n − 1. Such equations are called *recurrence relations* or *recurrences*.

Solve the recurrence relation $M(n)=M(n-1)+1$,i.e.,to find an explicit formula for $M(n)$ in terms of n only.

To determine a solution uniquely, we need an initial condition that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

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

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

$M(0) = 0$.

the calls stop when $n = 0$ ——↑ ↑—— no multiplications when $n = 0$

Thus, the recurrence relation and initial condition for the algorithm's number of multiplications $M(n)$:

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

Method of backward substitutions

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

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

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

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

$=M(n-3)+3$

…

$=M(n-i)+i$

…

=M(n −n)+n

=n.

Therefore *M(n)=n*

LECTURE 5:

EXAMPLE 2: consider educational workhorse of recursive algorithms: the Tower of Hanoi puzzle. We have n disks of different sizes that can slide onto any of three pegs. ConsiderA (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

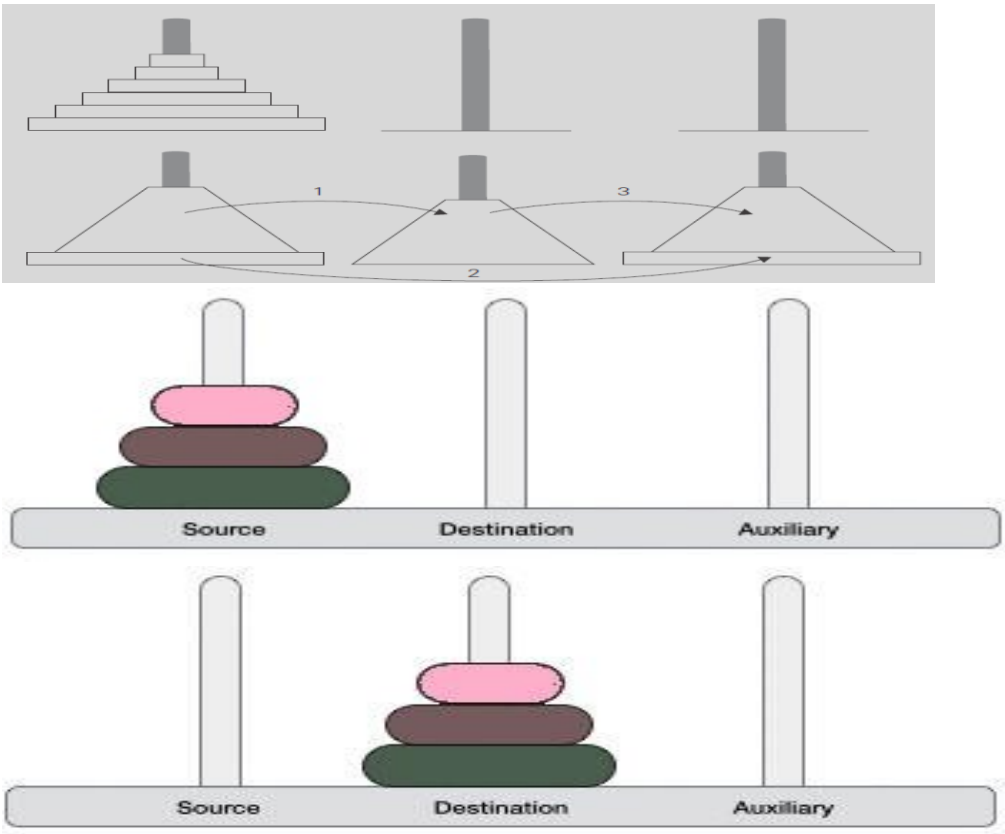


FIGURE 1.8 Recursive solution to the Tower of Hanoi puzzle

ALGORITHM TOH(n, A, C, B)

```
//Move disks from source to destination recursively
//Input: n disks and 3 pegs A, B, and  C
//Output: Disks moved to destination as in the source order.
if n=1
        Move disk from A to C
else
        Move top n-1 disks from A to B using C
        TOH(n - 1, A, B, C)
        Move top n-1 disks from B to C using A
        TOH(n - 1, B, C, A)
```

Algorithm analysis

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 $M(n)$:

$$M(n)=2M(n-1)+1 \text{ for } n>1,\ M(1)=1.$$

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

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

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

$=2^2M(n-2)+2+1$ sub. $M(n-2)=2M(n-3)+1$

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

$=2^3M(n-3)+2^2+2+1$ sub. $M(n-3)=2M(n-4)+1$

$=2^4M(n-4)+2^3+2^2+2+1$

…

$=2^iM(n-i)+2^{i-1}+2^{i-2}+\ldots+2+1=2^iM(n-i)+2^i-1.$

…

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$,

$M(n)=2^{n-1}M(n-(n-1))+2^{n-1}-1=2^{n-1}M(1)+2^{n-1}-1=2^{n-1}+2^{n-1}-1=2^n-1. \in \Theta(2^n)$

Thus, we have an exponential time algorithm

EXAMPLE 3: An investigation of a recursive version of the algorithm which finds the number of binary digits in the **binary representation** of a positive decimal integer.

ALGORITHM *BinRec(n)*

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

Algorithm analysis

The number of additions made in computing *BinRec*(Ln/2]) is *A*(Ln/2]), plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence $A(n) = A(\text{Ln}/2]) + 1$ for $n > 1$.

Since the recursive call send when n is equal to1 and there are no additions made then, the initial condition is $A(1)=0$.

The standard approach to solving such a recurrence is to solve it only for $n = 2^k$

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

$A(2^0)=0$.

Backward substitutions

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

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

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

. . .

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

. . .

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

Thus, we endup with $A(2^k)=A(1)+k= k$, or, after returning to the original variable $n=2^k$and hence $k = \log_2 n$,

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

EXAMPLE 3: An investigation of a recursive version of the algorithm which finds the number of binary digits in the **binary representation** of a positive decimal integer.

ALGORITHM *BinRec(n)*

//Input: A positive decimal integer *n*

//Output: The number of binary digits in *n*'s binary representation

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

else return *Bin Rec*(Ln/2])+1

Algorithm analysis

The number of additions made in computing *Bin Rec*(Ln/2]) is *A*(Ln/2]), plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence *A(n)* = *A(*Ln/2]*)* + 1 for *n* >1.

Since the recursive calls end when *n* isequalto1 and there are no additions made

then, the initial condition is *A(1)*=0.

The standard approach to solving such are currence is to solve it only for $n = 2^k$ $A(2^k) = A(2^{k-1}) + 1$ for $k > 0$,

$A(2^0)=0$.

Backward substitutions

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

. . .

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

. . .

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

Thus, we endup with $A(2^k)=A(1)+k=k$, or ,after returning to the original variable $n=2^k$and hence $k = \log_2 n$,

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

LH4 Review Questions

1.what is recurrence relations?

2.Solve the tower of Hanoi recurrence, using backward substitution

3.Solve the following recurrence relation?

T(n) = 7T(n/2) + 3n^2 + 2

What is Brute force?

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.
As an example, consider the exponentiation problem: compute an for a nonzero number a and a nonnegative integer n.By the definition of exponentiation,
a^n = _a ∗……∗a_ {n times}
This suggests simply computing anby multiplying 1 by a n times.

The brute-force approach should not be overlooked as an important algorithm design strategy.
First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems. In fact, it seems to be the only general approach for which it is more difficult to point out problems it cannot tackle.

Second, 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.

Third, 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.

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

Finally, 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.

LECTURE 6:

1.6 BRUTE FORCE

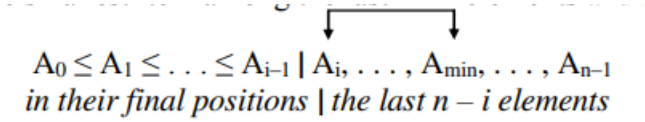
Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.
Selection Sort, Bubble Sort, Sequential Search, String Matching, Depth-First Search and Breadth-First Search, Closest-Pair and Convex-Hull Problems can be solved by Brute Force.
Examples:
1. Computing an : a * a * a * … * a (n times)
2. Computing n! : The n! can be computed as n*(n-1)* … *3*2*1
3. Multiplication of two matrices : C=AB
4. Searching a key from list of elements (Sequential search)

Advantages:
1. Brute force is applicable to a very wide variety of problems.
2. It is very useful for solving small size instances of a problem, even though it is inefficient.
3. The brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size for sorting, searching, and string matching.

Selection Sort
 First scan the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list.
 Then scan the list, starting with the second element, to find the smallest among the last n − 1 elements and exchange it with the second element, putting the second smallest element in its final position in the sorted list.
 Generally, on the ith pass through the list, which we number from 0 to n − 2, the algorithm searches for the smallest item among the last n − i elements and swaps it with Ai:

Selection Sort

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last n −1 elements and exchange it with the second element, putting the second smallest element in its final position. Generally, on the ith pass through the list, which we number from 0 to n −2, the algorithm searches
for the smallest item among the last n –i elements and swaps it with Ai :



$A_0 \le A_1 \le \ldots \le A_{i-1} \mid A_i, \ldots, A_{min}, \ldots, A_{n-1}$
in their final positions | the last n – i elements

After n −1 passes, the list is sorted.
Here is pseudocode of this algorithm, which, for simplicity, assumes that the
list is implemented as an array:

ALGORITHM SelectionSort(A[0..n − 1])
//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 nondecreasing 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*]

89	45	68	90	29	34	**17**
17	| 45	68	90	**29**	34	89
17	29	| 68	90	45	**34**	89
17	29	34	| 90	**45**	68	89
17	29	34	45	| 90	**68**	89
17	29	34	45	68	| 90	**89**
17	29	34	45	68	89	| 90

The sorting of list 89, 45, 68, 90, 29, 34, 17 is illustrated with the selection sort algorithm

The analysis of selection sort is straightforward. The input size is given by the number of elements n; the basic operation is the key comparison A[j]<A[min]. The number of times it is executed depends only on the array size and is given by the following sum:

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

Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs.
Note: The number of key swaps is only $\Theta(n)$, or, more precisely n – 1.

LH6 Review Questions:

1.What is Selection sort?
2.Write an algorithn for selection sort
3.what is brute force?

LECTURE 7:

1.7 Bubble Sort:

The bubble sorting algorithm is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after n − 1 passes the list is sorted. Pass i ($0 \le i \le n-2$) of bubble sort can be represented by the following:

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \le \ldots \le A_{n-1}$$

ALGORITHM BubbleSort(A[0..n − 1])

//Sorts a given array by bubble sort

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

//Output: Array A[0..n − 1] sorted in nondecreasing order

for $i \leftarrow 0$ **to** n − 2 **do**

  for $j \leftarrow 0$ **to** $n - 2 - i$ **do**

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

The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated as an example.

89 ?↔ 45 68 90 29 34 17
45 89 ?↔ 68 90 29 34 17
45 68 89 ?↔ 90 ?↔ 29 34 17
45 68 89 29 90 ?↔ 34 17
45 68 89 29 34 90 ?↔ 17
45 68 89 29 34 17 | 90

45 ?↔ 68 ?↔ 89 ?↔ 29 34 17 | 90
45 68 29 89 ?↔ 34 17 | 90
45 68 29 34 89 ?↔ 17 | 90
45 68 29 34 17 | 89 90

etc.

etc.

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort

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

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

$$C_{worst}(n) \in \Theta(n^2)$$

LECTURE 8:

1.8 SEQUENTIAL SEARCH

The general searching problem: it is called sequential search . To repeat, the algorithm simply 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). A simple extra trick is often employed in implementing sequential search: if we append the search key to the end of the list, the search for the key will have to be successful, and therefore we can eliminate the end of list check altogether. Here is pseudocode of this enhanced version.

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

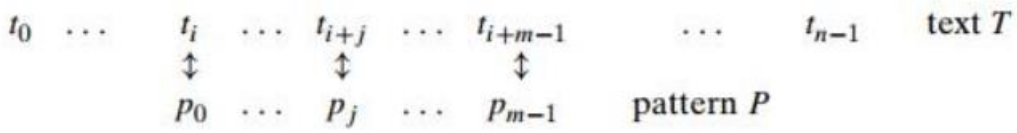
```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is
//        equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
        i ← i + 1
if i < n return i
else return −1
```

Another straightforward improvement can be incorporated in sequential search if a given list is known to be sorted: searching in such a list can be stopped as soon as an element greater than or equal to the search key is encountered.

1.9 Brute Force String Matching

given a string of n characters called the ***text*** and a string of m characters $(m \le n)$ 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

that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:



If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

Brute-force: Scan text LR, compare chars, looking for pattern,

Step 1 Align pattern at beginning of text
Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until

- **all characters are found to match (successful search); or**
- **a mismatch is detected**

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

An operation of the algorithm is illustrated in Figure 3.3. Note that for this example, the algorithm shifts the pattern almost always after a single character comparison. The worst case is much worse: 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. (Problem 6 in this section's exercises asks you to give a specific example of such a situation.) Thus, in the worst case, the algorithm makes

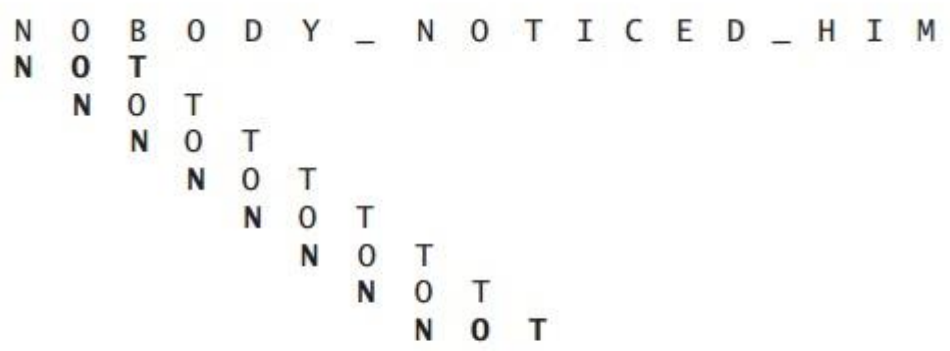


FIGURE 3.3 Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

$m(n - m + 1)$ character comparisons, which puts it in the $O(nm)$ class. For a typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons (check the example again). Therefore, the average-case efficiency should be considerably better than the worst-case efficiency. Indeed it is: for searching in random texts, it has been shown to be linear, i.e., $\Theta(n)$.