

# UNIT-III

## ALGORITHMS INDUCTION AND RECURSION

- An algorithm is a finite sequence of precise instructions for performing a computation or for solving a problem.

The term algorithm is a corruption of the name al-khowarizmi, a mathematician of the ninth century, whose book on Hindu numerals is the basis of modern decimal notation. Originally the word algorithm was used for the rules for performing arithmetic using decimal notation.

Algorithm evolved into the word algorithm by the eighteenth century, with the growing interest in computing machines, the concept of an algorithm was given a more general meaning to arithmetic.

In this book, we will discuss algorithms that solve a wide variety of problems. In this section we will use the problem of finding the largest integer in a finite sequence of integers to illustrate the concept of an algorithm and the properties algorithms have.

Example:

Describe an algorithm for finding the maximum value in a finite sequence of integers.

1. Set the temporary maximum equal to the first integer in the sequence.

(int The temporary maximum will be the largest integer examined at any stage of the procedure.)

2. Compare the next integer in the sequence to the temporary maximum. and if it is larger than the temporary maximum. set the temporary maximum equal to this integer.

3. Repeat the previous step if there are more integers in the sequence.

4. Stop when there are no integers left in the sequence. The temporary maximum at the greatest integer in the sequence.

algorithm | finding the maximum element in a finite

procedure max ( $a_1, a_2, \dots, a_n$ );

$max := a_1$ ,

for  $i := 2$  to

if  $max < a_i$  then  $max := a_i$

return  $max$  {max is the largest.}

This algorithm first assigns the initial term of the sequence,  $a_1$ , to the variable max. The "for" loop is used to successively examine terms of the sequence. If a term is greater than the current value of max, it is assigned to be the new value of max. The algorithm terminates after all terms have been examined. The value of max on termination is the maximum element in the sequence.

To gain insight into how an algorithm works it is useful to construct a trace that shows its steps when given specific input. For instance, a trace of Algorithm 1 with input 8, 4, 11, 3, 10 begins with the algorithm setting max to 8, the value of the initial term. It then compares 4, the second term, with 8, the current value of max. Because  $4 \leq 8$ , max is unchanged. Next the algorithm compares the third term, 11, with 8, the current value of max. Because  $8 < 11$ , max is set equal to 11. The algorithm then compares 3, the fourth term, and 11, the current value of max. Because  $3 \leq 11$ , max is unchanged. Finally, the algorithm compares 10, the first term, and 11, the current value of max. As  $10 \leq 11$ , max remains unchanged. Because there are five terms we have  $n=5$ . So after examining 10, the last term, the algorithm terminates, with  $\text{max}=11$ . When it terminates, the algorithm reports that 11 is the largest term in the sequence.

## Properties of algorithms

There are several properties that algorithms generally share. They are useful to keep in mind when algorithms are described.

These properties are:

1. Input: An algorithm has input values from a specified set,

2. Output: from each set of input values an algorithm produces output values from a specified set.

The output values are the solution to the problem.

Definiteness:- An algorithm should produce the correct output. The steps of an algorithm must be defined precisely

correctness: An algorithm should produce the correct output values for each set of input values.

finiteness: An algorithm should produce the desired output after a finite number of steps for any input in the set.

Effectiveness: It must be possible to perform each step of an algorithm exactly and in a finite amount of time.

Generality: The procedure should be applicable for all problems of the desired form, not just for a particular set of input values.

Example 2:-

198

show that Algorithm I for finding the maximum element in a finite sequence of integers has all the properties listed.

Solution:- The input to Algorithm I is a sequence of integers. The output is the largest integer in the sequence. Each step of the algorithm is precisely defined, because only assignments, a finite loop, and conditional statements occur. To show that the algorithm is correct, we must show that when the algorithm terminates, the value of the variable max equals the maximum of the terms of the sequence. To see this, note that the initial value of max is the first term of the sequence: as successive terms of the sequence are examined, max is updated to the value of a term if the term exceeds the maximum of the terms previously examined. This argument shows that when all the terms have been examined, max equals the value of largest term. The algorithm uses a finite number of steps, because it terminates after all the integers in the sequence have been examined. The algorithm can be carried out in a finite amount of time because each step is either a comparison or an assignment; there are a finite number of these steps, and each of these two operations takes a finite amount of time. Finally, Algorithm I is general, because it can be used to find the maximum of any finite sequence of integers.

# Searching Algorithms

69

The problem of locating an element in an ordered list occurs in many contexts. For instance, a program that checks the spelling of words searches for them in a dictionary, which is just an ordered list of words. Problems of this kind are called searching problems. We will discuss several algorithms in section 3.3.

The general searching problem can be described as follows: Locate element  $x$  in a list of distinct elements  $a_1, a_2, \dots, a_n$ , or determine that it is not in the list. The solution to this search problem is the location of the term in the list that equals  $x$  (that is,  $i$  is the solution if  $x=a_i$ ) and  $0$  if  $x$  is not in the list.

## The Linear Search:-

The first algorithm that we will present is called the linear or sequential search algorithm. The linear search algorithm begins by comparing  $x$  and  $a_1$ . When  $x=a_1$ , the solution is the location of  $a_1$  namely  $1$ . When  $x \neq a_1$ , compare  $x$  with  $a_2$ . If this process, comparing  $x$  successively with each term of the list until a match is found, where the solution is the location of that term. Unless no match occurs.

If the entire list has been searched without locating  $x$ , the solution is 0. The pseudocode for the linear search algorithm is displayed as Algorithm 2.

Procedure linearsearch ( $x$ ; integer,  $a_1, a_2, \dots, a_n$   
distinct)

$i := 1$

while ( $i \leq n$  and  $x \neq$

$i := i + 1$

if  $i \leq n$  then location :=

else location :=

return location {location is the subscript of the term that equals  $x$ , or is 0 if  $x$  is not found}

The Binary Search:

We will now consider another searching algorithm. The algorithm can be used when the list has terms occurring in order of increasing size.

This second searching algorithm is called the binary search algorithm. It proceeds by comparing the element to be located to the middle term of the list. The list is then split into two smaller sublists of the same size, or where one of these smaller lists has one fewer term than the other the search continues by restricting the search to the appropriate sublist based on the comparison of the element to be located and the middle term.

To search for 19 in the list.

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22.

first split this list, which has 16 terms, into two smaller lists with eight terms each, namely

1 2 3 5 6 7 8 10      12 13 15 16 18 19 20 22.

$10 < 19$ .

12 13 15 16      18 19 20 22.

$16 < 19$

18 19      20 22

$= 19$

We now specify the steps of the binary search algorithm. To search for the integer  $x$  in the list  $a_1, a_2, \dots, a_n$  where  $a_1 < a_2 < \dots < a_n$  begin by comparing  $x$  with the middle term  $a_m$  of the list, where  $m = \lceil (n+1)/2 \rceil$ . If  $x > a_m$  the search for  $x$  is restricted to the second half of the list, which is  $a_{m+1}, a_{m+2}, \dots, a_n$ , if  $x$  is not greater than  $a_m$ , the search for  $x$  is restricted to the first half of the list, which is  $a_1, a_2, \dots, a_n$ .

Using the same procedure, compare  $x$  to the middle term of the restricted list. Then restrict the search to the first or second half of the list. Repeat this process until a list with one term is obtained.

## Binary search algorithm

procedure binary search ( $x$ : integer,  $a_1, a_2 \dots a_n$ :  
increasing)

$i := 1$  { $i$  is left endpoint of search}

$j := n$  { $j$  is right endpoint of search}

while  $\neq$

$m := (i + j) / 2$

if  $x > a_m$  then  $i := m + 1$

else  $j :=$

if  $x = a_i$  then location :=

else location :=

return location {location is the subscript  $i$  of the  
term  $a_i$  equal to  $x$ , or 0 if  $x$  is not

This algorithm proceeds by successively  
narrowing down the part of the sequence being  
searched. At any given stage only the terms from  
 $a_i$  to  $a_j$  are under consideration. In other words,  $i$  and  
 $j$  are the smallest and largest subscripts of the  
remaining terms, respectively. Algorithm continues  
narrowing the part of the sequence being searched  
until only one term of the sequence remains, when  
this is done, a comparison is made to see whether  
this term equals  $x$ .

## Sorting :-

Suppose that we have a list of elements of a set. furthermore, suppose that we have a way to order elements of the set.

Sorting is putting these elements into a list in which the elements are in increasing order. for instance, sorting the list 7, 2, 1, 4, 5, 9 produces the list 1, 2, 4, 5, 7, 9 sorting the list d, h, i, c, a, f produces the list a, c, d, f, h.

There are many reasons why sorting algorithms interest computer scientists and mathematicians. Among these reasons are that some algorithms are easier to implement, some algorithms are more efficient, some algorithms are particularly clever. In this section we will introduce two sorting algorithms, the bubble sort and the insertion sort. Two other sorting algorithms, the selection sort and binary insertion sort, are introduced in the exercises, and the shaker sort is introduced in the supplementary Exercises.

## The Bubble Sort

54

The bubble sort is one of the simplest sorting algorithms, but not one of the most efficient. It puts a list into increasing order by successively comparing adjacent elements, interchanging them if they are in the wrong order. To carry out the bubble sort, we perform the basic operation, that is, interchanging a larger element with a smaller one following it, starting at the beginning of the list; for a full pass, we iterate this procedure until the sort is complete. Pseudocode for the bubble sort is given as Algorithm 4. We can imagine the elements in the list placed in a column. In the bubble sort, the smaller elements "bubble" to the top as they are interchanged with larger elements. The larger elements "sink" to the bottom. This is illustrated in Example 4.

Example 4. Use the bubble sort to put 3, 2, 4, 1, 5 into increasing order.

The steps of this algorithm are illustrated in figure 1. Begin by comparing the first two elements, 3 and 2. Because  $3 > 2$ , interchange 3 and 2, producing the list 2, 3, 4, 1, 5. Because  $3 < 4$ , continue by comparing 4 and 1. Because  $4 > 1$ , interchange 1 and 4, producing the list 2, 3, 1, 4, 5. Because  $4 < 5$ , the first pass is complete. The first pass guarantees that the largest element, 5, is in the correct position.

The second pass begins by comparing 2 and 3. Because these are in the correct order, 3 and 1 are compared. Because  $3 > 1$ , these numbers are interchanged, producing 2, 1, 3, 4, 5. Because for this  $3 < 4$ , these numbers are in the correct order. It is not necessary to do any more comparisons for this pass because 5 is already in the correct position. The second pass guarantees that the two largest elements, 4 and 5, are in their correct positions.

55

The third pass begins by comparing 2 and 1. These are interchanged because  $2 > 1$ , producing 1, 2, 3, 4, 5. Because  $2 < 3$  these two elements are in the correct order. It is not necessary to do any more comparisons for this pass because 4 and 5 are already in the correct positions. The third pass guarantees that the three largest elements, 3, 4, and 5, are in their correct positions.

The fourth pass consists of one comparison, namely, the comparison of 1 and 2. Because  $1 < 2$ , these elements are in the correct order. This completes the bubble sort.

#### first pass

3	2	2	2
2	↑3	3	3
4	↓4	↑4	1
1	1	↓1	↑4
5	5	5	↑5

Second pass

2	2	2
3	3	1
1	1	3
4	4	4
5	5	5

Third pass

2	1
1	2
3	3
4	4
5	5

fourth pass

- $\frac{1}{2}$  : an interchange
- $\frac{3}{4}$  : pair in correct order
- $\frac{5}{ }$  : numbers in coloy
- $\frac{}{ }$  guaranteed to be in correct order

Bubble sort algorithm :-

```

procedure bubblesort (a1, ..., an: real numbers with n)
  for i := 1 to n -
    for j := 1 to n -
      if ai > aj+1 then interchange ai and aj+1
  {a1, ..., an is in increasing}

```

## THE INSERTION SORT

57

The insertion sort is a simple sorting algorithm, but it is usually not the most efficient. To sort a list with  $n$  elements, the insertion sort begins with the second element. The insertion sort compares this second element with the first element and inserts it before the first element if it does not exceed the first element if it exceeds the first element. At this point, the first two elements are in the correct order. The third element is then compared with the first element, and if it is larger than the first element, it is compared with the second element; it is inserted into the correct position among the first three elements.

### Example 5

Use the insertion sort to put the elements of the list 3, 2, 4, 1, 5 in increasing order.

Solution: The insertion sort first compares 2 and 3. Because  $3 > 2$ , it places 2 in the first position, producing the list 2, 3, 4, 1, 5. At this point, 2 and 3 are in the correct order. Next, it inserts the third element, 4, into the already sorted part of the list by making the comparisons  $4 > 2$  and  $4 > 3$ . Because  $4 > 3$ , 4 remains in the third position. At this point, the list is 2, 3, 4, 1, 5 and we know that the ordering of the first three elements is correct. Next, we find the correct place for the fourth element, 1, among the already sorted elements, 2, 3, 4. Because  $1 < 2$  we obtain the list 1, 2, 3, 4, 5. Finally, we insert 5 into the correct position.

position by successively comparing it to 1, 2, 3 and 4. Because  $5 > 4$ , it stays at the end of the list, producing the correct order for the entire list.

Insertion sort algorithm

procedure insertion sort ( $a_1, a_2, \dots, a_n$ : real numbers with  $n$

for  $j := 2$  to

$i := 1$

while  $a_j >$

$i := i + 1$

$m := a_i$

for  $k := 0$  to  $j - i - 1$

$a_{j-k} := a_{j-k-1}$

$a_i := m$

string matching

naive string matcher:-

The input to this algorithm is the pattern we wish to match,  $P = P_1, P_2, \dots, P_m$ , and the text  $T = t_1, t_2, \dots, t_n$ . When this pattern begins at position  $s+1$  in the text  $T$ , we say that  $P$  occurs with shift  $s$  in  $T$ . That is, when  $t_{s+1} = P_1, t_{s+2} = P_2, \dots, t_{s+m} = P_m$ . To find all valid shifts, the naive string matching runs through all possible shifts from  $s=0$  to  $s=n-m$ , checking whether  $s$  is a valid shift.

## Naive String Algorithm

procedure string match ( $n, m$ : positive integers,  
 $m \leq n, t_1, t_2, \dots, t_n, P_1, P_2, \dots, P_m$ ;

for  $s := 0$  to  $n -$

$\quad j := 1$

while [ $j \leq m$  and  $t_{s+j} =$

$\quad \quad j := j + 1$

if  $j > m$  then print "s is a valid.

$s = 0$

e c e y e y e  
 $\downarrow$   
 eye

$s = 3$

c y e y e y e  
 $\downarrow$   
 eye

$s = 4$

e c e y e y e  
 $\downarrow$   
 eye

## Cashier's Algorithm.

procedure change( $c_1, c_2, \dots, c_r$ ; values of denominations of coins,

$c_1 > c_2 > \dots > c_r$ ;  $n$ : a positive  
for  $i := 1$  to

$d_i := 0$  { $d_i$  counts the coins of denomination  
while  $n \geq i$ }

$d_i := d_i + 1$  {add a coin of denomination  
 $n := n - c_i$

{ $d_i$  is the number of coins of denomination  $c_i$  in the  
change for  $i = 1, 2$ .

We have described the cashier's algorithm, a greedy algorithm for making change, using any finite set of coins with denominations  $c_1, c_2, \dots, c_r$ . In the particular case where the four denominations are quarters, dimes, nickels, and pennies, we have  $c_1 = 25, c_2 = 10, c_3 = 5$ , and  $c_4 = 1$ . For this case, we will show that this algorithm leads to an optimal solution in the sense that it uses the fewest coins possible. Before we embark on our proof, we show that there are sets of coins for which the cashier's algorithm does not necessarily produce change using the fewest coins possible.

## Greedy Algorithm :-

67

procedure schedule ( $s_1 \leq s_2 \leq \dots \leq s_n$ : start times of  
 $e_1 \leq e_2 \leq \dots \leq e_n$ : ending times of  
sort talks by finish time and reorder so that  
 $s := \emptyset$        $e_1 \leq e_2 \leq \dots$   
for  $j := 1$  to  
    if talk  $j$  is compatible w/s.  
         $s := s \cup \{ \text{talk } j \}$   
return  $s$ .

## The Growth of functions

### Introduction.

The Time required to solve a problem depends on more than only the number of operations it uses. The time also depends on the hardware and software used to run the program that implements the algorithm. However, when we change the hardware and software used to implement an algorithm, we can closely approximate the time required to solve a problem of size  $n$  by multiplying the previous time required by a constant. For example. On a supercomputer we might be able to solve the problem of size  $n$  a million times faster than we can on a PC.

52

However, this factor of one million will not depend on  $n$ . One of the advantages of using big-O notation, which we introduce in this section, is that we can estimate the growth of a function without worrying about constant multipliers or smaller order terms. This means that, using big-O notation, we do not have to worry about the hardware and software used to implement an algorithm. Furthermore, using big-O notation, we do not have to worry about the hardware and software used to implement an algorithm. Furthermore, using big-O notation, we can assume that the different operations used in an algorithm take the same time, which simplifies the analysis considerably.

### Big-O notation

The growth of functions is often described using a special notation. Definition 1 describes this notation.

Let  $f$  and  $g$  be functions from the set of integers or the set of real numbers to the numbers. We say that  $f(x)$  is  $O(g(x))$  if there are constants  $c$  and  $k$ .

$$|f(x)| \leq c|g(x)|$$

whenever  $x > k$  (This is read as  $x$  is big-oh of

working with the definition of bigo notation

63

$f(x) \leq c(g(x))$  for  $x > k$ . This approach is illustrated in Example.

Example 1 Show that  $f(x) = x^2 + 2x + 1$  is  $O(x^2)$ .

Solution:-

We observe that we can readily estimate the size of  $f(x)$  when  $x > 1$  because  $x < x^2$  and  $1 < x$  when  $x > 1$ . It follows that

$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

Whenever  $x > 1$ , as shown in figure 1, consequently, we can take  $c = 4$  and  $k = 1$  as witness to show that  $f(x)$  is  $O(x^2)$ . That is,  $f(x) = x^2 + 2x + 1 \leq 4x^2$  whenever  $x > 1$ .

Alternatively, we can estimate the size of  $f(x)$  when  $x > 2$ . When  $x > 2$ , we have  $2x \leq x^2$  and  $1 \leq x^2$ . Consequently, if  $x > 2$  we have

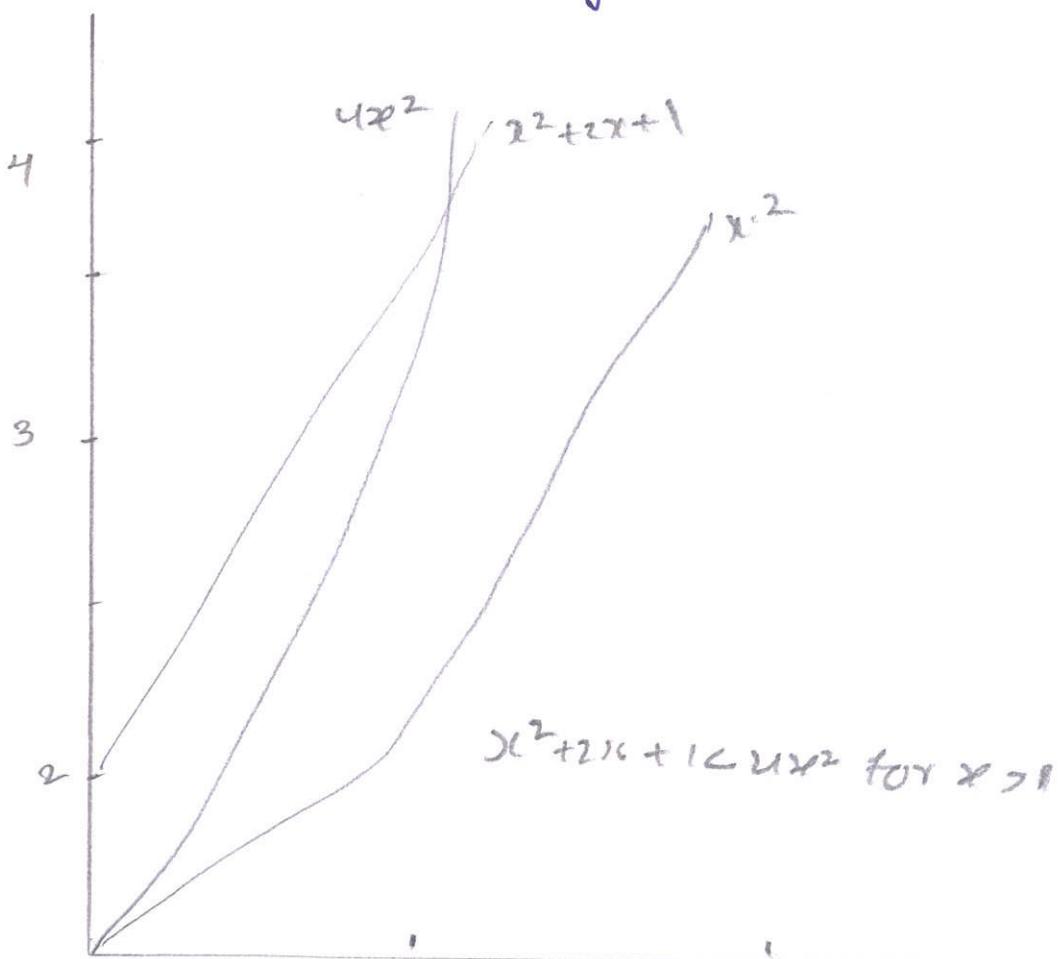
$$0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2.$$

It follows that  $c = 3$  and  $k = 2$  are also witness to the relation  $f(x)$  is  $O(x^2)$ .

Observe that in the relationship  $f(x)$  is  $O(x^2)$ .  
 $x^2$  can be replaced by any function that has larger values than  $x$  for all  $x \geq k$  for some positive real number  $k$ . For example,  $f(x)$  is  $O(x)$ ,  $f(x)$  is  $O(x^2+x+7)$ , and so on.

It is also true that  $x$  is  $O(x^2+2x+1)$ , because  $x \leq x^2+2x+1$  whenever  $x \geq 1$ . This means that  $c=1$  and  $k=1$  are witnesses to the relationship  $x^2$  is  $O(x^2+2x+1)$ .

Note that in Example 1 we have two functions,  $f(x) = x^2+2x+1$  and  $g(x) = x^2$ , such that  $f(x)$  is  $O(g(x))$  and  $g(x)$  is  $O(f(x))$  the latter fact following from the inequality  $x^2 \leq x^2+2x+1$ , which holds for all nonnegative real numbers  $x$ ; we say that functions



Example 1 The function  $x^2+2x+1$  is  $O(x^2)$ . 63

$f(x)$  and  $g(x)$  that satisfy both of these big-O relationships are of the same order. we will return to this notation later in this section

Remark: The fact that  $f(x)$  is  $O(g(x))$  is sometimes written  $f(x)=O(g(x))$ . However, the equals sign in this notation does not represent a genuine equality rather, this notation tells us that an inequality holds relating the values of the functions  $f$  and  $g$  for sufficiently large numbers in the domains of these functions. However, it is acceptable to write  $f(x) \in O(g(x))$  because  $O(g(x))$  represents the set of functions that are  $O(g(x))$ .

When  $f(x)$  is  $O(g(x))$ , and  $h(x)$  is a function that has larger absolute values than  $g(x)$  does for sufficiently large values of  $x$ , it follows that  $f(x)$  is  $O(h(x))$ . In other words, the function  $g(x)$  in the relationship  $f(x)$  is  $O(g(x))$  can be replaced by a function with larger absolute values. To see this, note that if

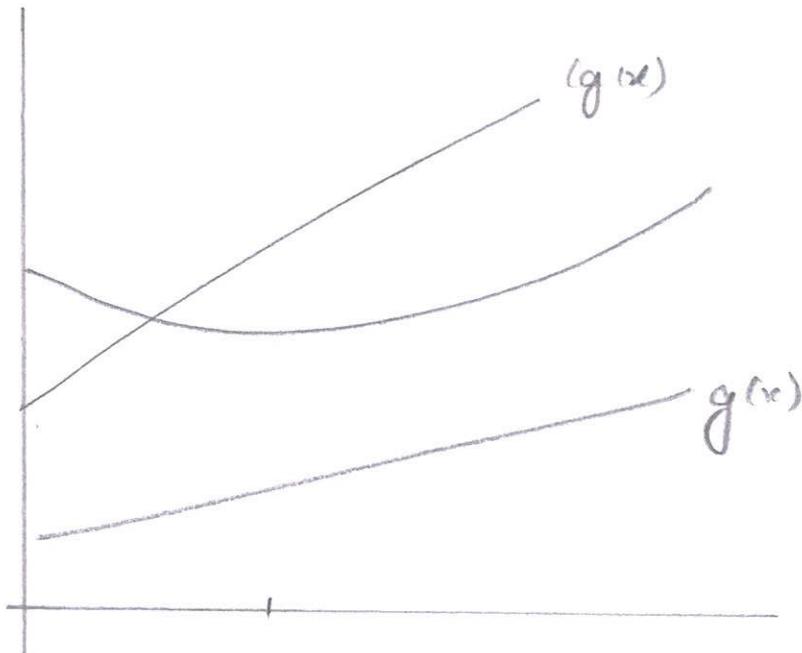
$|f(x)| \leq c|g(x)|$  if  $x > k$ ,  
 and if  $|h(x)| > |g(x)|$  for all  $x > k$ , then

$$|f(x)| \leq c|h(x)| \text{ if } x > k.$$

hence,  $f(x)$  is  $O(h(x))$ .

When big-O notation is used, the function  $g$  in the relationship  $f(x)$  is  $O(g(x))$  is often chosen to have the smallest growth rate of the functions belonging to a set of reference functions, such as functions of the form  $x^n$ , where  $n$  is a positive real number.

In subsequent discussions, we will almost always deal with functions that take on only positive values. All references to absolute value can be dropped when working with big-O estimates for such functions. Figure 2 illustrates the relationship  $f(x)$  is  $O(g(x))$



The part of the graph of  $f(x)$  that satisfies  $f(x) < g(x)$  is shown in gray.

The function  $f(x)$  is  $O(g(x))$ .

Theorem 1 :

Let  $f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ ,  
where  $a_0, a_1, \dots, a_{n-1}, a_n$  are real numbers  
 $f(x)$  is

proof :

Using the triangle inequality, if  $x > 1$  we have

$$\begin{aligned}
 |f(x)| &= |a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0| \\
 &\leq |a_n| |x^n| + |a_{n-1}| |x^{n-1}| + \dots + |a_1| |x| + |a_0| \\
 &= x^n (|a_n| + |a_{n-1}| x + \dots + |a_1| x^{n-1} + |a_0| x^n) \\
 &\leq x^n (|a_n| + |a_{n-1}| + \dots + |a_1| + |a_0|).
 \end{aligned}$$

This shows that

$$|f(n)| \leq (x)^n$$

where  $c = |a_n| + |a_{n-1}| + \dots + |a_0|$  whenever  $x \geq 1$

Hence, the witnesses  $c = |a_n| + |a_{n-1}| + \dots + |a_0|$  and  $k = 1$  show that  $f(n)$  is

We now give some examples involving functions that have the set of positive integers as their domains.

### Example 6

Give big-O estimates for the factorial function and the logarithm of the factorial function, where the factorial function  $f(n) = n!$  is defined by

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

whenever  $n$  is a positive integer, and  $0! = 1$ , for example,

$$1! = 1, \quad 2! = 1 \cdot 2 = 2, \quad 3! = 1 \cdot 2 \cdot 3 = 6.$$

$$4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$$

Note that the function  $f$  grows rapidly. For instance

$$20! = 2,432,902,008,176,640,000$$

89

solution: A big-O estimate for  $n!$  can be obtained by noting that each term in the product does not exceed  $n$ . Hence.

$$\begin{aligned} n! &= 1 \cdot 2 \cdot 3 \cdot \dots \cdot n \\ &\leq n \cdot n \cdot n \cdot \dots \cdot n \\ &= n^n. \end{aligned}$$

This inequality shows that  $n!$  is  $O(n^n)$ , taking  $c=1$  and  $k=1$  as witness. Taking logarithms of both sides of the inequality established for  $n!$ , we obtain

$$\log n! \leq \log n^n = n \log n.$$

This implies that  $\log n!$  is  $O(n \log n)$ , again taking  $c=1$  and  $k=1$  as witnesses.

## The Growth of combinations of functions

many algorithms are made up of two or more separate subprocedures. The number of steps used by a computer to solve a problem with input of a specified size using such an algorithm is the sum of the number of steps used by these subprocedures. To give a big-O estimate for the number of steps needed, it is necessary to find big-O estimates for the number of steps used by each subprocedure and then combine these estimates.

Big-O estimates of combinations of functions can be provided if care is taken when different big-O estimates are combined. In particular, it is often necessary to estimate the growth of the sum and the product of two functions. What can be said if big-O estimates for each of two functions are known? to see what sort of estimates hold for the sum and the product of two functions. suppose that  $f_1(n)$  is  $O(g_1(x))$  and  $f_2(x)$  is  $O(g_2(x))$ .

from the definition of big-O notation, there are constants  $c_1, l_2, k_1$  and  $k_2$  such that

$$|f_1(x)| \leq c_1 g_1(x)$$

When  $x > k_1$  and

$$|f_2(x)| \leq (2 \lg_2(x))$$

when  $x > k_2$ , To estimate the sum of  $f_1(x)$  and  $f_2(x)$ , note that

$$|(f_1 + f_2)(x)| = |f_1(x)| + |f_2(x)|$$

$$(|f_1(x)| + |f_2(x)|)_{x < 1}$$

when  $x$  is greater than both  $k_1$  and  $k_2$ , it follows from the inequalities for  $|f_1(x)|$  and  $|f_2(x)|$  that

$$\begin{aligned} |f_1(x)| + |f_2(x)| &\leq (1 \lg_1(x)) + (2 \lg_2(x)) \\ &\leq (1(g(x)) + (2 \lg(x))) \\ &= ((+1) \lg(x)) \\ &= (1 \lg(x)). \end{aligned}$$

where  $c = c + c$  and  $g(x) = \max(g_1(x), g_2(x))$ .

[Here  $\max(a, b)$  denotes the maximum, or larger, of  $|a|$  and  $|b|$ .]

This inequality shows that  $(f_1 + f_2)(x) \leq (c g(x))$  whenever  $x > k$ , where  $k$  where  $k = \max(k_1, k_2)$ , we state this useful result as Theorem 2.

Suppose that  $f_1(x)$  is  $O(g_1(x))$  and that  $f_2(x)$  is  $O(g_2(x))$ . Then  $(f_1 + f_2)(x)$  is  $O(g(x))$ .

$$g(n) = (\max(|g_1(x)|, |g_2(x)|)) \text{ for all } x$$

We often have big estimates for  $f_1$  and  $f_2$  in terms of the same function  $g$ . In this situation, theorem 2 can be used to show that  $(f_1 + f_2)(x)$  is also  $O(g(x))$ , because  $\max(g(x), g(n)) = g(x)$  this result is stated in corollary 1.

Suppose that  $f_1(x)$  and  $f_2(x)$  are both  $O(g(n))$ . Then  $(f_1 + f_2)(x)$  is  $O(g(x))$ .

In a similar way big-O estimates can be derived for the product of the function  $f_1$  and  $f_2$  when  $x$  is greater than  $\max(k_1, k_2)$  it follows that

$$\begin{aligned} |(f_1 f_2)(x)| &= |f_1(x)| |f_2(x)| \\ &\leq \epsilon_1 |g_1(x)| \epsilon_2 |g_2(x)| \\ &\leq c |g_1 g_2(x)|, \end{aligned}$$

where  $c = \epsilon_1 \epsilon_2$ . From this inequality, it follows that  $f_1(x) f_2(x)$  is  $O(g_1 g_2(x))$ , because there are constants  $c$  and  $k$  namely,  $c = 1$  (and  $k = \max(k_1, k_2)$ ) such that  $|f f(x)| \leq c g_1(x) g_2(x)$  whenever  $x > k$ .

# Complexity of Algorithms

73

When does an algorithm provide a satisfactory solution to a problem? First, it must always produce the correct answer. How this can be demonstrated will be discussed in chapter.

One measure of efficiency is the time used by a computer to solve a problem using the algorithm, when input values are of a specified size. A second measure is the amount of computer memory required to implement the algorithm when input values are of a specified size.

An analysis of the time required to implement the algorithm to solve a problem of a particular size involves the time complexity of the algorithm. An analysis of the computer memory required involves the space complexity of the algorithm. Considerations of the time and space complexity of an algorithm are essential when algorithms are implemented. It is important to know whether an algorithm will produce an answer in a microsecond, a minute, or a billion years.

## Time complexity

74

The time complexity of an algorithm can be expressed in terms of the number of operations used by the algorithm when the input has a particular size. The operations used to measure time complexity can be the comparison of integers, the addition of integers, the multiplication of integers, the division of integers, or any other basic operation.

Time complexity is described in terms of the number of operations required instead of actual computer time because of the difference in time needed for different computers to perform basic operations. Moreover, it is quite complicated to break all operations down to the basic bit operations that a computer uses, furthermore, the fastest computers in existing two bits in  $10^{-11}$  second (10 picoseconds), but personal computers may require  $10^{-8}$  second (10 nanoseconds), which is 1000 times as long, to do the same operations.

## Complexity of Matrix Multiplication

To definition of the product of two matrices can be expressed as an algorithm for computing the product of two matrices, suppose that  $C = [c_{ij}]$  is the  $m \times n$  matrix that is the product of the  $m \times k$  matrix  $A = [a_{ij}]$  and the  $k \times n$  matrix  $B = [b_{ij}]$ . The algorithm based on the definition of the matrix product is expressed in pseudocode in Algorithm.

### Algorithm Matrix Multiplication:-

Procedure matrix multiplication(A,B):

for  $i := 1$  to

    for  $j := 1$  to

$c_{ij} := 0$

        for  $q := 1$  to  $k$

$c_{ij} := c_{ij} + a_{iq} b_{qj}$

return  $C$  {  $C = [c_{ij}]$  is the product of  $A$  and

we can determine the complexity of this algorithm in terms of the number of additions and multiplications used.

# Understanding the complexity of algorithms.

33

Table I displays some common terminology used to describe the time complexity of algorithms. For example, an algorithm that finds the largest of the first 100 terms of a list of  $n$  elements by applying Algorithm 1 to the sequence of the first 100 terms, where  $n$  is an integer with 100, has constant complexity because it uses 99 comparisons no matter what  $n$  is. The linear search algorithm has linear complexity and the binary search algorithm has logarithmic complexity. Many important algorithms have  $n \log n$ , or linearithmic complexity.

complexity	Terminology
$\Theta(1)$	constant complexity
$\Theta(\log n)$	logarithmic complexity
$\Theta(n)$	linear complexity
$\Theta(n \log n)$	linearithmic complexity
$\Theta(n^b)$	polynomial complexity
$\Theta(b^n)$ where $b > 1$	exponential complexity
$\Theta(n!)$	factorial complexity