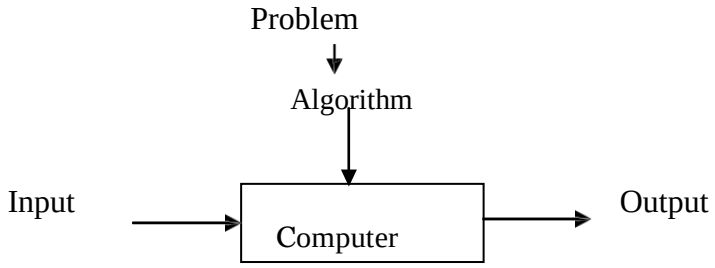
UNIT – I

ALGORITHM

An algorithm is a sequence of unambiguous instruction for solving a problem, for obtaining a required output for any legitimate input in a finite amount of time.



FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

There are two kinds of efficiency

- **Time efficiency** - indicates how fast an algorithm in question runs.
- **Space efficiency** - deals with the extra space the algorithm requires.

MEASURING AN INPUT SIZE

An algorithm's efficiency as a function of some parameter n indicating the algorithm's input size. In most cases, selecting such a parameter is quite straightforward.

For example, it will be the size of the list for problems of sorting, searching, finding the list's smallest element, and most other problems dealing with lists.

For the problem of evaluating a polynomial $\mathbf{p(x) = a_n x^n + \ldots + a_0}$ of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by one than its degree.

There are situations, of course, where the choice of a parameter indicating an input size does matter.

Example - computing the product of two n-by-n matrices.

There are two natural measures of size for this problem.

- the matrix order n.
- the total number of elements n in the matrices being multiplied.

Since there is a simple formula relating these two measures, we can easily switch from one to the other, but the answer about an algorithm's efficiency will be qualitatively different depending on which of the two measures we use.

The choice of an appropriate size metric can be influenced by operations of the algorithm in question. For example, how should we measure an input's size for a spell-checking algorithm? If the algorithm examines individual characters of its input, then we should measure the size by the number of characters; if it works by processing

words, we should count their number in the input.

We should make a special note about measuring size of inputs for algorithms involving properties of numbers (e.g., checking whether a given integer n is prime).

For such algorithms, computer scientists prefer measuring size by the number b of bits in the n's binary representation:

$$b = \lfloor \log_2 n \rfloor + 1$$

This metric usually gives a better idea about efficiency of algorithms in question.

WORST CASE, BEST CASE AND AVERAGE CASE EFFICIENCES

It is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algorithm's input.

But there are many algorithms for which running time depend not only on an input size but also on the specifics of a particular input.

Example: sequential search.

This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

Here is the algorithm's pseudo code, in which, for simplicity, a list is implemented as an array. (It also assumes that the second condition A[i] i= K will not be checked if the first one, which checks that the array's index does not exceed its upper bound, fails.)

ALGORITHM Sequential Search(A[0..n -1], K)

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n -1] and a search key K
//Output: Returns the index of the first element of A that matches K
//          or -1 if there 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.

Worst case efficiency

The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.

In this, when there are no matching elements or the first matching element

happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n:

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

The way to determine is quite straightforward. To analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count C(n) among all possible inputs of size n and then compute this worst- case value $C_{worst}(n)$

The worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size n, the running time will not exceed $C_{worst}(n)$ its running time on the worst-case inputs.

Best case Efficiency

The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size.

Analyze the best case efficiency as follows,

First, determine the kind of inputs for which the count C (n) will be the smallest among all possible inputs of size n. (Note that the best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.)

Then ascertain the value of C (n) on these most convenient inputs. Example- for sequential search, best-case inputs will be lists of size n with their first elements equal to a search key; accordingly, $C_{best}(n) = 1$.

The analysis of the best-case efficiency is not nearly as important as that of the worst- case efficiency. But it is not completely useless. For example, there is a sorting algorithm (insertion sort) for which the best-case inputs are already sorted arrays on which the algorithm works very fast. Thus, such an algorithm might well be the method of choice for applications dealing with almost sorted arrays.

Average case efficiency

It yields the information about an algorithm about an algorithm‘s behavior on a typical and random input.

To analyze the algorithm's average-case efficiency, must make some assumptions about possible i*n*puts of size n.

The investigation of the average case efficiency is considerably more difficult than investigation of the worst case and best case efficiency.

It involves dividing all instances of size n .into several classes so that for each instance of the class the number of times the algorithm's basic operation is executed is the same.

Then a probability distribution of inputs needs to be obtained or assumed so that the expected value of the basic operation's count can then be derived.

The average number of key comparisons $C_{avg}(n)$ can be computed as follows,

- Let us consider again sequential search. The standard assumptions are,
- In the case of a successful search, the probability of the first match occurring in the ith position of the list is pin 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 is n with the probability of such a search being (1 - p).

Therefore,

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

$$= \frac{p}{n} [1 + 2 + 3 + \ldots. + i + \ldots. + n] + n (1 - p)$$

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

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

Example,

If p = 1 (i.e., the search must be successful), the average number of key comparisons made by sequential search is (n + 1) /2.

If p = 0 (i.e., the search must be unsuccessful), the average number of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

ASYMPTOTIC NOTATIONS

Step count is to compare time complexity of two programs that compute same function and also to predict the growth in run time as instance characteristics changes. Determining exact step count is difficult and not necessary also. Because the values are not exact quantities. We need only comparative statements like $c_1n^2 \leq t_p(n) \leq c_2n^2$.

For example, consider two programs with complexities $c_1n^2 + c_2n$ and c_3n respectively. For small values of n, complexity depend upon values of c_1, c_2 and c_3. But there will also be an n beyond which complexity of c_3n is better than that of $c_1n^2 + c_2n$.This value of n is called break-even point. If this point is zero, c_3n is always faster (or at least as fast). Common asymptotic functions are given below.

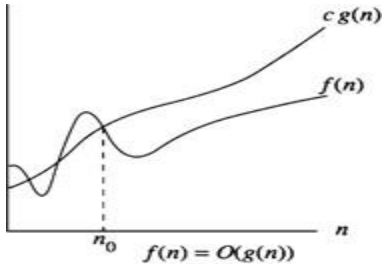
Function	Name
1	Constant
n	Linear
log n	Logarithmic
n log n	Logarithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

Asymptotic function in ascending order,

$1<\log n<\sqrt{n}<n<n\log n<n^2<n^3<\ldots<2^n<3^n<\ldots<n^n$

Big'Oh'Notation(O)

O(g(n)) = { f(n) : there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$ }. It is the upper bound of any function. Hence it denotes the worse case complexity of any algorithm. We can represent it graphically as



Find the Big Oh for the following functions:

Linear Functions

Example 1

$f(n) = 3n + 2$

General form is $f(n) \le cg(n)$

When $n \ge 2$, $3n + 2 \le 3n + n = 4n$

Hence $f(n) = O(n)$, here $c = 4$ and $n_0 = 2$

When $n \ge 1$, $3n + 2 \le 3n + 2n = 5n$

Hence $f(n) = O(n)$, here $c = 5$ and $n_0 = 1$

Hence we can have different c, n_0 pairs satisfying for a given function.

Example 2

$f(n) = 3n + 3$

When $n \geq 3$, $3n + 3 \leq 3n + n = 4n$

Hence $f(n) = O(n)$, here $c = 4$ and $n_0 = 3$

Example 3

$f(n) = 100n + 6$

When $n \geq 6$, $100n + 6 \leq 100n + n = 101n$

Hence $f(n) = O(n)$, here $c = 101$ and $n_0 = 6$

Quadratic Functions

Example 1

$f(n) = 10n^2 + 4n + 2$

When $n \geq 2$, $10n^2 + 4n + 2 \leq 10n^2 + 5n$

When $n \geq 5$, $5n \leq n^2$, $10n^2 + 4n + 2 \leq 10n^2 + n^2 = 11n^2$ Hence $f(n) = O(n^2)$, here $c = 11$ and $n_0 = 5$

Example 2

$f(n) = 1000n^2 + 100n - 6$

$f(n) \leq 1000n^2 + 100n$ for all values of n.

When $n \geq 100$, $5n \leq n^2$, $f(n) \leq 1000n^2 + n^2 = 1001n^2$ Hence $f(n) = O(n^2)$, here $c = 1001$ and $n_0 = 100$

Exponential Functions

Example

$f(n) = 6*2^n + n^2$ When $n \geq 4$,

$n^2 \leq 2^n$

So $f(n) \leq 6*2^n + 2^n = 7*2^n$

Hence $f(n) = O(2^n)$, here $c = 7$ and $n_0 = 4$

Constant Functions

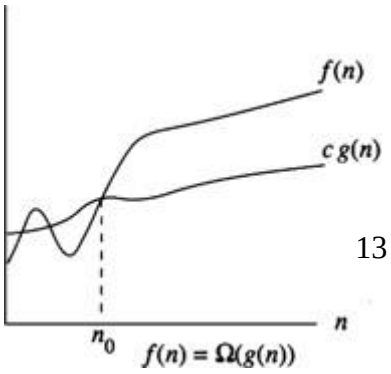
Example

$f(n) = 10$

$f(n) = O(1)$, because $f(n) \leq 10*1$

Omega Notation(Ω)

Ω (g(n)) = { f(n) : there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ }. It is the lower bound of any function. Hence it denotes the best case complexity of any algorithm. We can represent it graphically as



13

Example

f(n) = 3n + 2

3n + 2 > 3n for all n.

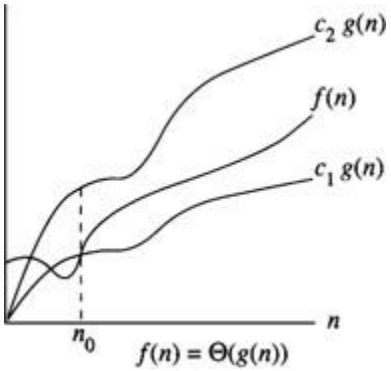
Hence f(n) = Ω(n)

Similarly we can solve all the examples specified under Big _Oh'.

Theta Notation(Θ)

Θ(g(n)) = {f(n) : there exist positive constants c_1,c_2 and n_0 such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$ }

If f(n) = Θ(g(n)), all values of n right to n_0 f(n) lies on or above $c_1g(n)$ and on or below $c_2g(n)$. Hence it is asymptotic tight bound for f(n)



Little-O Notation

For non-negative functions, f(n) and g(n), f(n) is little o of g(n) if and only if f(n) =O(g(n)), but f(n) ≠ Θ(g(n)). This is denoted as "f(n) = o(g(n))".

This represents a loose bounding version of Big O. g(n) bounds from the top, but it does not bound the bottom.

Little Omega Notation

For non-negative functions, f(n) and g(n), f(n) is little omega of g(n) if and only if f(n) = Ω(g(n)), but f(n) ≠ Θ(g(n)). This is denoted as "f(n) = ω(g(n))".

Much like Little Oh, this is the equivalent for Big Omega. g(n) is a loose lower boundary of the function f(n); it bounds from the bottom, but not from the top.

CONDITIONAL ASYMPTOTIC NOTATION

Many algorithms are easier to analyze if initially restrict the attention to instances whose size satisfies a certain condition, such as being a power of 2. Consider, for example, the divide and conquer algorithm for multiplying large integers that we saw in the Introduction. Let *n* be the size of the integers to be multiplied.

The algorithm proceeds directly if n=1, which requires ‘a’ microseconds for an appropriate constant ‘a’. If n>1, the algorithm proceeds by multiplying four pairs of integers of size n/2 (or three if we use the better algorithm).Moreover, it takes a linear amount of time to carry out additional tasks. For simplicity, let’s say that the additional work takes at most ‘bn’ microseconds for an appropriate constant ‘b’.

PROPERTIES OF BIG-OH NOTATION

Following are the properties of asymptotic notations:-

General Property

- If f(n) = Θ(g(n)), then a * f(n) = Θ(g(n))
- If f(n) = O(g(n)), then a * f(n) = O(g(n))
- If f(n) = Ω(g(n)), then a * f(n) = Ω(g(n))

Example:

$f(n)= 2n^2+3$

$f(n) = O(g(n))$

$f(n) = O(n^2)$ then

$4 * f(n) = 4 * (2n^2+3)$

$g(n) = O(n^2)$

Transitive

- If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$
- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
- If $f(n) = o(g(n))$ and $g(n) = o(h(n))$, then $f(n) = o(h(n))$
- If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$
- If $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$, then $f(n) = \omega(h(n))$

Example:

$f(n) =n$

$g(n)=n^2$

$h(n)=n^3$

If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$

$f(n) = O(n^2)$ and $g(n) = O(n^3)$, then $f(n) = O(n^3)$

Reflexivity

- $f(n) = \Theta(f(n))$
- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$

Example

$f(n)=2n^2+3$

$f(n)= O(n^2)$

Symmetry

- $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$

Example:

$f(n)=n^2$, $g(n)=n^2$

$f(n) = \Theta(n^2)$ if and only if $g(n) = \Theta(n^2)$

Transpose Symmetry

- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
- $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

Example:

f(n)=n

g(n) = n^2

f(n) = O(g(n)) if and only if g(n) = Ω(f(n))

f(n) = $O(n^s)$ if and only if g(n) = Ω(n)

RECURRENCE

When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence

Recurrence Equation

A recurrence relation is an equation that recursively defines a sequence. Each term of the sequence is defined as a function of the preceding terms.

Example:

```
void test(int n)
{
  if(n>0)
  {
        printf("%d",n);
        test(n-1);
  }
}
```

The total call of the recursive function is n+1 times

f(n)=n+1

f(n)=O(n)

SOLVING RECURRENCE EQUATION

There are four methods for solving Recurrence:

- Substitution Method
- Iteration Method
- Recursion Tree Method
- Master Method

Substitution Method:

The Substitution Method consists of two main steps:

1. Guess the Solution.
2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.

Example:

```
   void test(int n)
{
     if(n>0)
    {
                printf("%d",n);
                test(n-1);
      }
}
```

T(n)=T(n-1)+1

$$T(n)=\begin{cases} 1 & n=0 \\ T(n-1)+1, & n>0 \end{cases}$$

T(n)=T(n-1)+1
T(n-1)=T(n-2)+1
T(n-2)=T(n-3)+1
.
.

Substitute T(n-1) in T(n)
T(n)=[T(n-2)+1]+1
T(n)=T(n-2)+2 (substitute T(n-2))
T(n)=T(n-3)+3
. (k times)
.
T(n)=T(n-k)+k
Assume n-k=0
n=k
Substitute n=k in T(n)
T(n)=T(n-n)+n
=T(0)+n T(0)=1
T(n)=1+n
f(n)=O(n)

Iteration Methods

It means to expand the recurrence and express it as a summation of terms of n and initial condition.

Example :

Consider the Recurrence

$T(n) = 1$ **if** $n=1$
$= 2T(n-1)$ **if** $n>1$

Solution:

$T(n) = 2T(n-1)$
$= 2[2T(n-2)] = 2^2T(n-2)$
$= 4[2T(n-3)] = 2^3T(n-3)$
$= 8[2T(n-4)] = 2^4T(n-4)$ (Eq.1)

Repeat the procedure for i times

$T(n) = 2^i T(n-i)$
Put $n-i=1$ or $i= n-1$ in (Eq.1)
$T(n) = 2^{n-1} T(1)$
$= 2^{n-1} .1$ {T (1) =1 .....given}
$= 2^{n-1}$

Recursion Tree Method

1. Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.

2. In general, we consider the second term in recurrence as root.

3. It is useful when the divide & Conquer algorithm is used.

4. It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represents the cost of a single sub-problem.

5. We sum the costs within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.

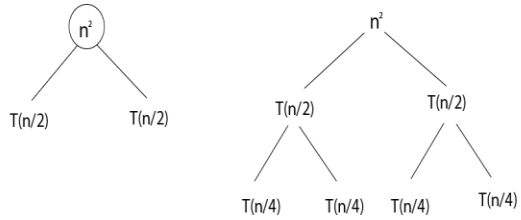
6. A Recursion Tree is best used to generate a good guess, which can be verified by the Substitution Method.

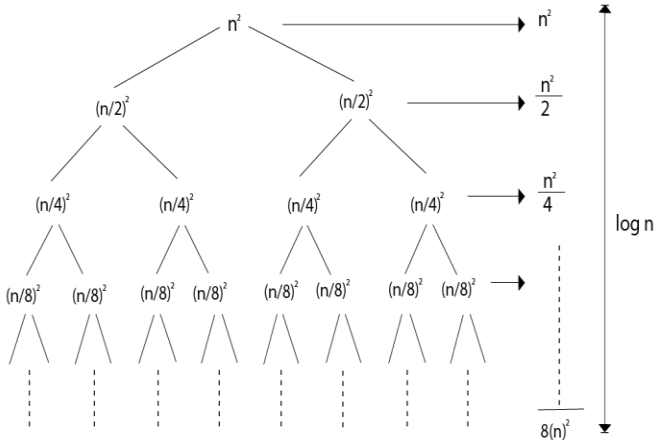
Example

Consider $T(n) = 2T(n/2) + n^2$

We have to obtain the asymptotic bound using recursion tree method.

Solution: The Recursion tree for the above recurrence is





$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots\ldots \text{log n times.}$$

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

$$\mathbf{T(n) = \theta n^2}$$

The master method

In the analysis of algorithms, the master theorem for divide-and-conquer recurrences provides an asymptotic analysis (using Big O notation) for recurrence relations of types that occur in the analysis of many divide and conquer algorithms.

The runtime of an algorithm such as the 'p' above on an input of size 'n', usually denoted T(n), can be expressed by the recurrence relation

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

In the function to the analysis of a recursive algorithm, the constants and function take on the following significance:

- n is the size of the problem.

- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- f (n) is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the subproblems.

It is possible to complete an asymptotic tight bound in these three cases:

- Compare Case 1: $n/(\log n)$ is not $O(n^{1-\varepsilon})$ for any $\varepsilon > 0$, since n^{ε} grows faster than $\log n$
- Case 2: $n/(\log n)$ is not $\Theta(n)$, since $(n)/(n/(\log n)) = \log n$
- Case 3: $n/(\log n)$ is not $\Omega(n^{1+\varepsilon})$ for any $\varepsilon > 0$, since $(n^{1+\varepsilon})/(n/(\log n)) = n^{\varepsilon}\log n$

Example

$T(n)=2T(n/2)+n$

Here, $a=2$, $b=2$, $\log_b a=\log_2 2=1$
Now, $n^{\log_b a}=n^{\log_2 2}=n$
Also, $f(n)=n$
So, $n^{\log_b a}=n=f(n)$
(comparing $n^{\log_b a}$ with $f(n)$ $=>f(n)=\Theta(n^{\log_b a})$
So, case 2 can be applied and thus $T(n)=\Theta(n^{\log_b a}\log n)=\Theta(n \log n)$

ANALYSIS OF LINEAR SEARCH

Linear Search, as the name implies is a searching algorithm which obtains its result by traversing a list of data items in a linear fashion. It will start at the beginning of a list, and mosey on through until the desired element is found, or in some cases is not found. The aspect of Linear Search which makes it inefficient in this respect is that if the element is not in the list it will have to go through the entire list.

Linear Search Steps:

Step 1 - Does the item match the value looking for?

Step 2 - If it does match, return found the item!

Step 3 - If it does not match, advance and repeat the process.
Step 4 - Reached the end of the list and still no value found? Well obviously the item is not in the list! Return -1 to signify that the value have not found

Search the value 7 from a list of elements. The indexes of the array of element have size of 5.

0	1	2	3	4	5
[30	12	20	7	10	42]

Start the search from the index 0, here the value is 30. It is not equal to 7.So move on to the next index.

In index 1, the value is 12 not equal to 7. Then move on the next index until found the value 7.

In index 3 it matches with 7. So the value is in index 3 and stops the search.

Algorithm

```
//linearSearch Function
int linearSearch(int data[], int length, int val)
{
  for (int i = 0; i <= length; i++)
  {
   if (val == data[i])
  {
        return i;
  }//end if
}//end for
   return -1;   //Value was not in the list
}//end linearSearch Function
```

Analysis & Conclusion

Worst Case:

The worse case for Linear Search is achieved if the element to be found is not in the list at all. This would entail the algorithm to traverse the entire list and return nothing. Thus the worst case running time is: O(n).

Average Case:

The average case is in short revealed by insinuating that the average element would be somewhere in the middle of the list or n/2. This does not change since we are dividing by a constant factor here, so again the average case would be: O(n).

Best Case:

The best case can be a reached if the element to be found is the first one in the list. This would not have to do any traversing spare the first one giving this a constant time complexity: O(1).

References:

1. Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007
2. K.S. Easwarakumar, Object Oriented Data Structures using C++, Vikas publishing House Pvt Ltd., 2000