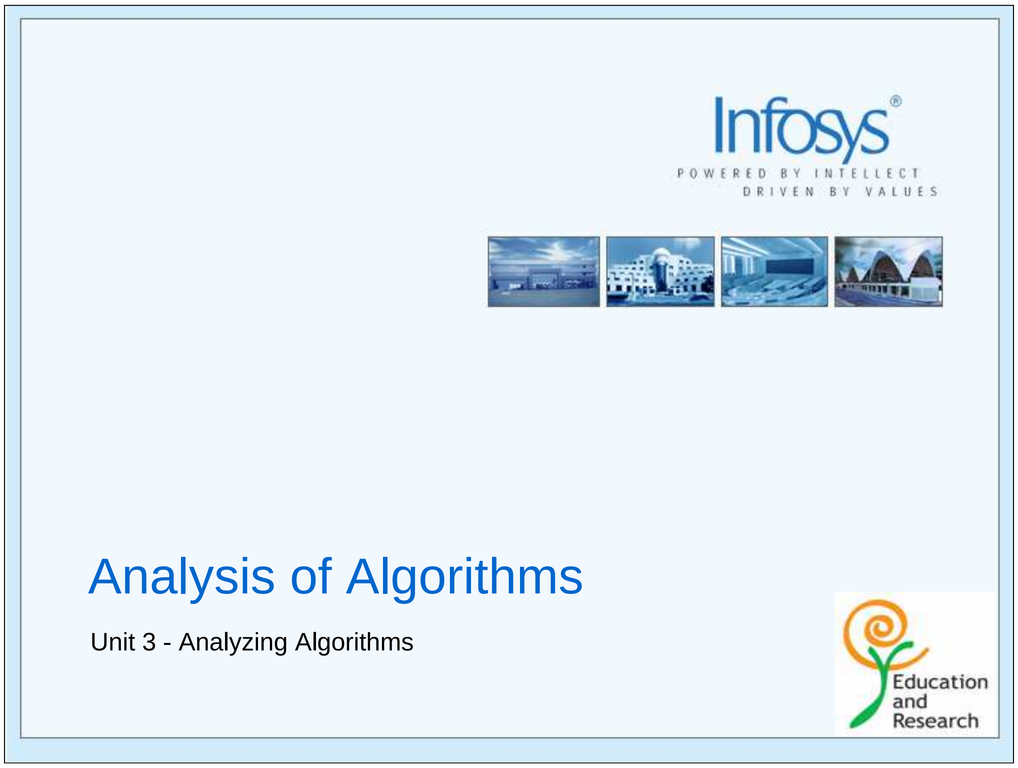
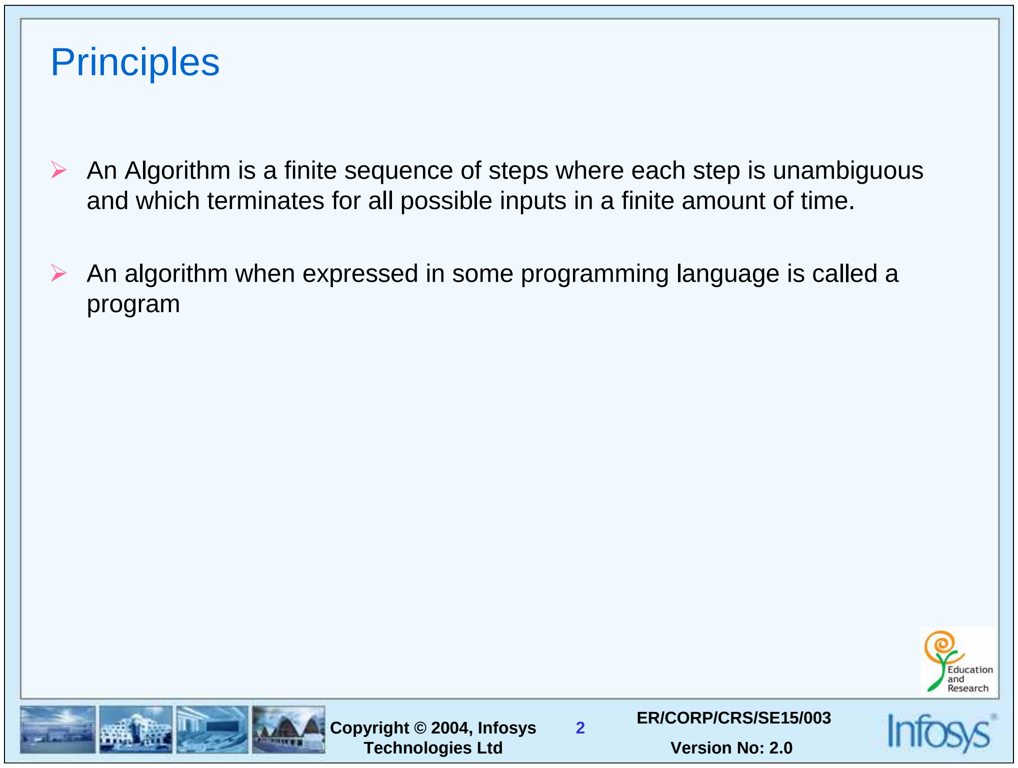


Analysis of Algorithms

Unit 3 - Analyzing Algorithms

In this chapter, we will:

- •Understand the strength of performing a machine independent analysis of a solution
- •Study the advantages of doing an ***“Apriori Analysis”*** when compared to a ***“Posterior Analysis”***
- •Understand the concept of ***input size*** and learn to perform an ***instruction count*** on the input size
- •Learn the concept of ***worst case*** and ***average complexity*** measures



Principles

- An Algorithm is a finite sequence of steps where each step is unambiguous and which terminates for all possible inputs in a finite amount of time.
- An algorithm when expressed in some programming language is called a program

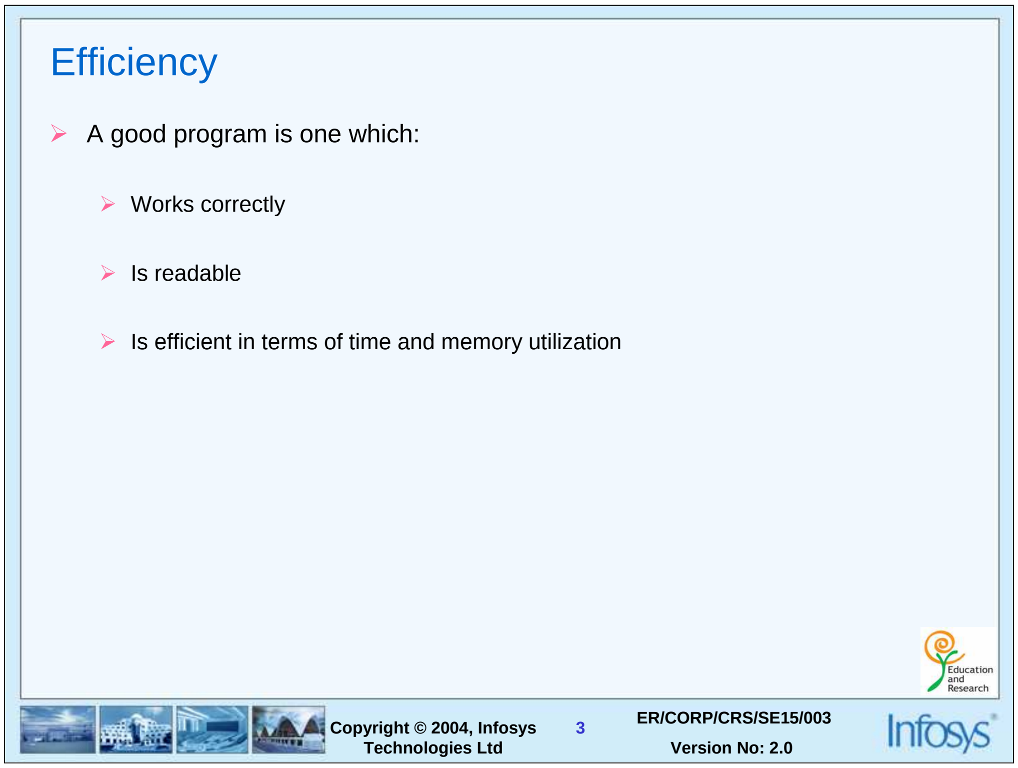
ER/CORP/CRS/SE15/003
Version No: 2.0

Principles:

•A solution for a given problem may be in the form of an algorithm or a program

•An algorithm is a step by step procedure for solving the given problem. An algorithm is independent of any programming language and machine. An algorithm is also called a *Pseudo Code*. Pseudo means false. An algorithm is called a false code as it is not tagged to any specific language syntax.

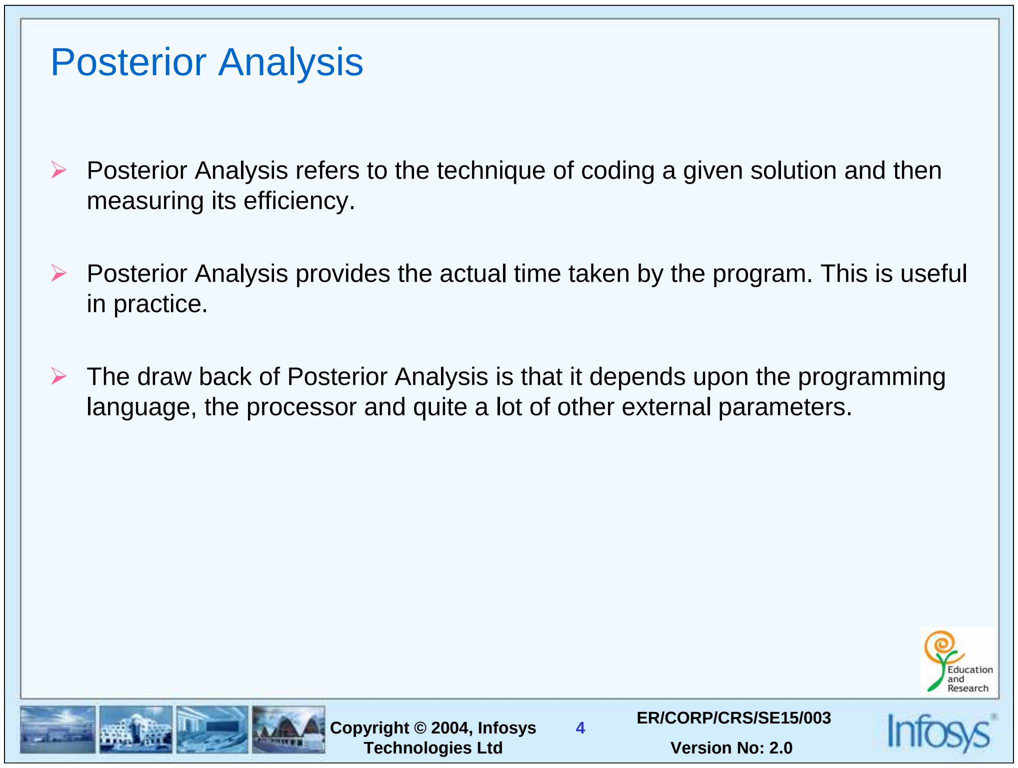
•A program is a language specific implementation of the algorithm. A program is synonymous with *Code*.



Efficiency

- A good program is one which:
  - Works correctly
  - Is readable
  - Is efficient in terms of time and memory utilization

As mentioned in chapter – 1, most of the software problems do not have a unique solution and given multiple solutions we are always interested in finding out the better one. First and foremost a good solution should give the desired output. It should also be readable so that we could understand the solution easily. Finally a good solution should give the desired output as soon as possible and must also utilize less resources.

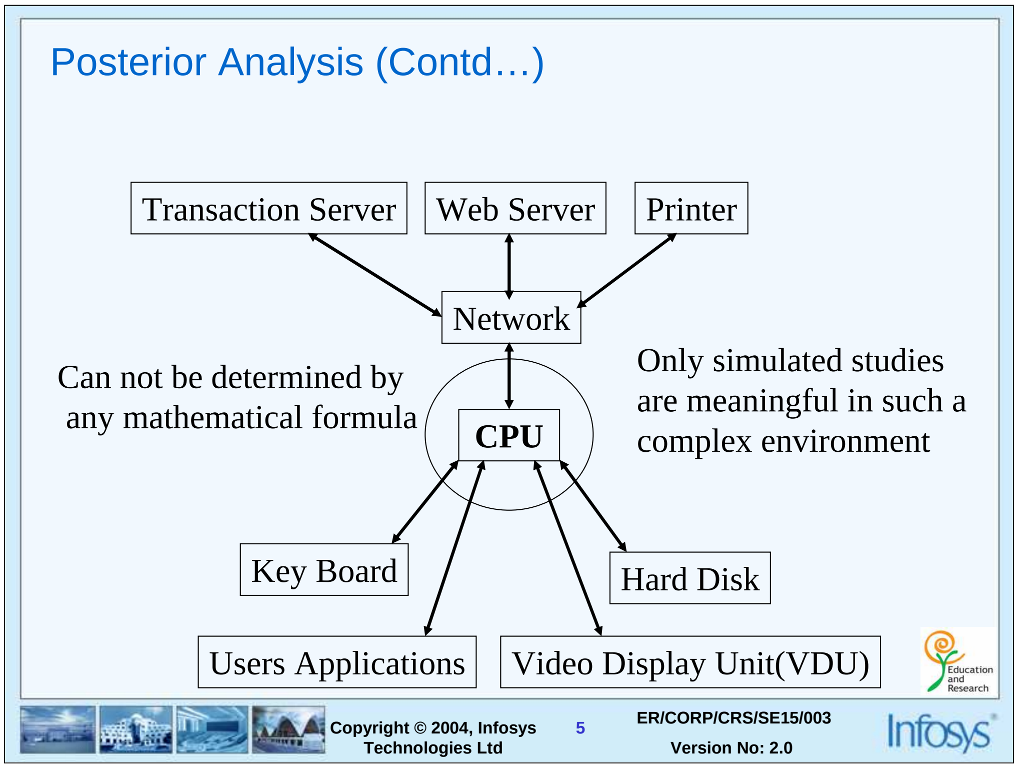


Posterior Analysis

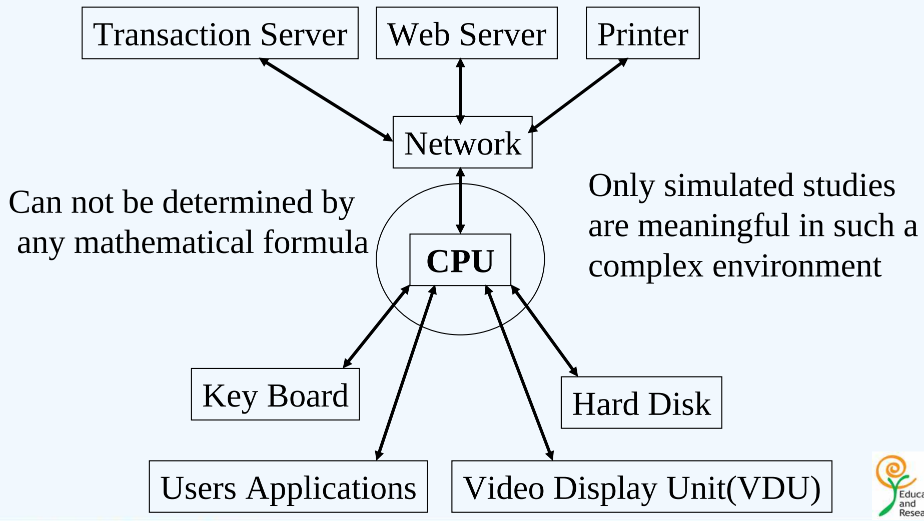
- Posterior Analysis refers to the technique of coding a given solution and then measuring its efficiency.
- Posterior Analysis provides the actual time taken by the program. This is useful in practice.
- The draw back of Posterior Analysis is that it depends upon the programming language, the processor and quite a lot of other external parameters.

•The efficiency of a solution can be found by implementing it as program and measuring the time taken by the code to give the desired output. The amount of memory occupied by the code can also be calculated. This kind of analysis is called Posterior Analysis. Posterior means *“later in time”* . The analysis is done after coding the solution and not at the design phase.

•We could do a posterior analysis for the different possible solutions of a given problem. The inherent draw back of posterior analysis is highlighted in the following slide.



Posterior Analysis (Contd…)



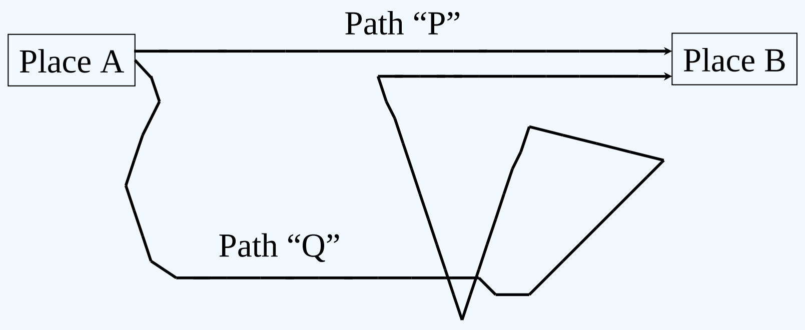
Can not be determined by any mathematical formula

Only simulated studies are meaningful in such a complex environment

•Consider a complex system as shown above. The efficiency in terms of a time taken by a program depends on the CPU's performance apart from other factors. Suppose a given problem has more than one solution and we need to find out the better one using posterior analysis. In such a complex system it is difficult to analyze these solutions practically because there could interrupts to the CPU when one of the solution is being analyzed in which case the different solutions may not be comparable.

•Then how do we compare two solutions for their efficiency?

Apriori Analysis

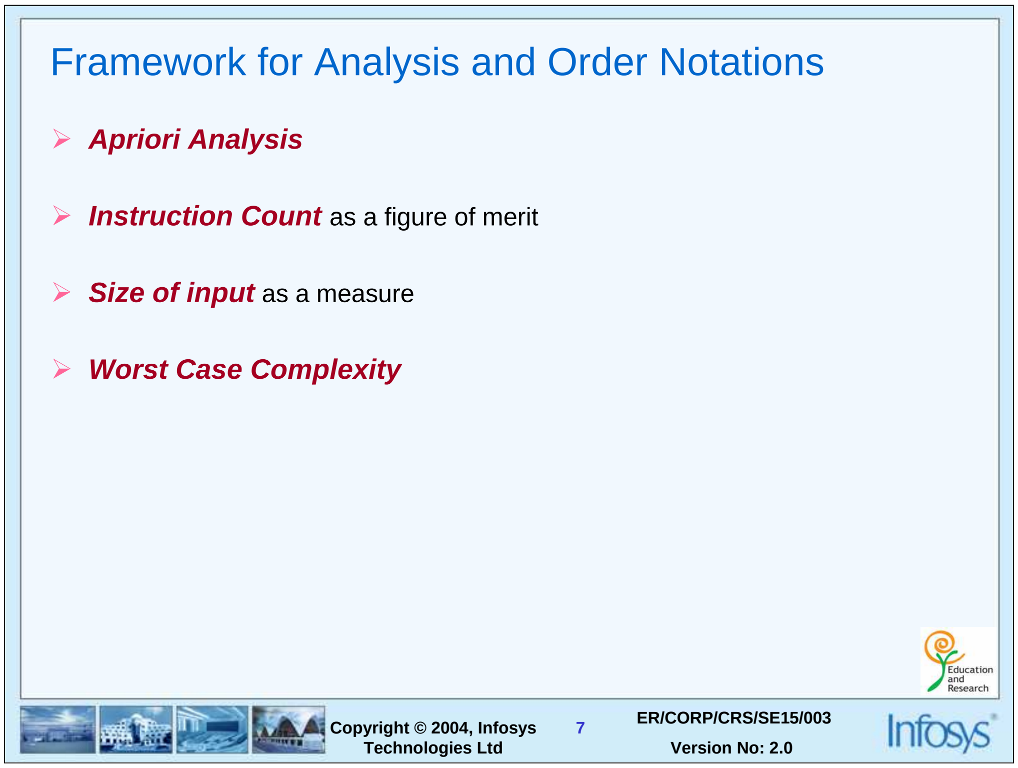


•Consider the above given example. Suppose that we need to go from place A to place B and there exists two paths P and Q for the same. It is clear that for any given method (in this physical example Method will comprises- vehicle, its speed etc.) Path P will always take less time.

•Similarly for a given problem in computer science the method would consist of the processor, main memory etc. and the algorithm will correspond to the path.

•So given two are more algorithms for a problem we are interested in doing an machine independent analysis of these algorithms and say algorithm A is better than the remaining algorithms.

•This leads to the principle of ***Apriori Analysis***. Apriori analysis means doing an analysis of the solutions before we code.



Framework for Analysis and Order Notations

- ***Apriori Analysis***
- ***Instruction Count*** as a figure of merit
- ***Size of input*** as a measure
- ***Worst Case Complexity***

•***Donald Knuth***, the famous Computer Scientist, provided the ***Foundation for Analysis of Algorithms***.

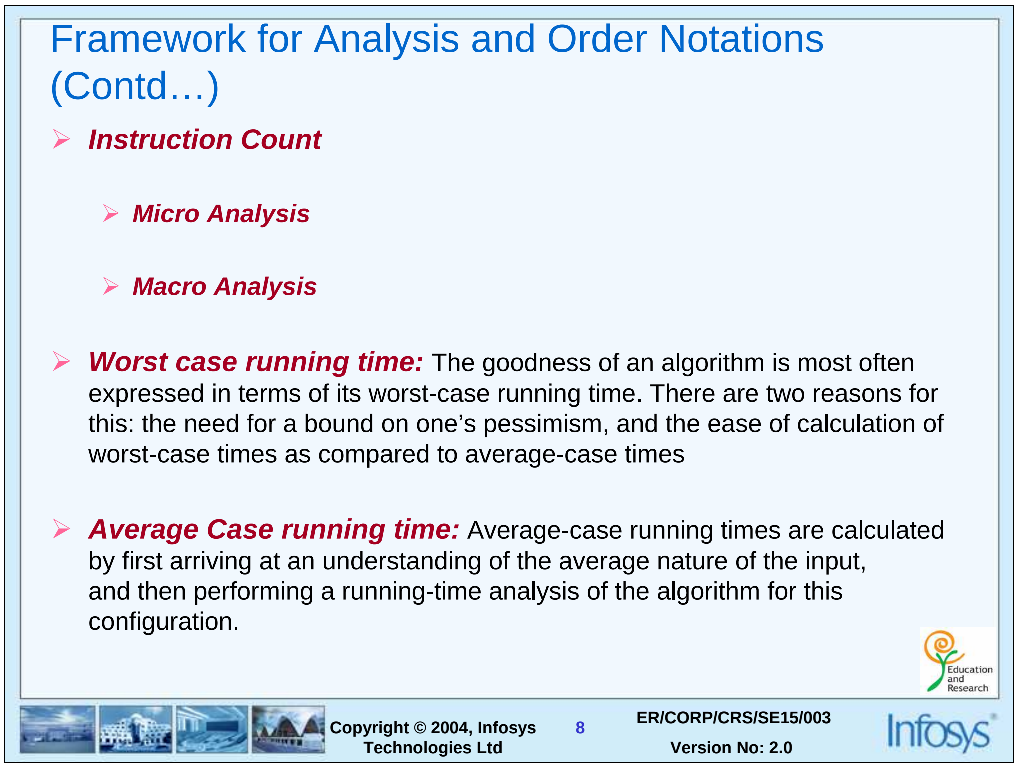
•He gave the principles and tools for analyzing algorithms. The principle of *Apriori Analysis* was introduced by Donald Knuth.

•To do an Apriori Analysis we need some tools. The tools include the *instruction count* and *size of input*.

•To do an apriori analysis we need to identify the basic operations (or instructions) in the algorithm and do a count (*Instruction Count*) on these operations. The instruction count is used as a figure of merit in doing the analysis.

•The *size of input* (*input size*) is used as a measure to quantify the input.

•Worst Case Complexity refers to the maximum instruction count among all instruction counts. Let $\mathbf{I_n}$ denote the set of all inputs of size **n** and let **IC(i)** denote the instruction count on a specific input **i**. Worst Case Complexity, **WCC = Max IC(i), for all I in $\mathbf{I_n}$**



Framework for Analysis and Order Notations (Contd…)

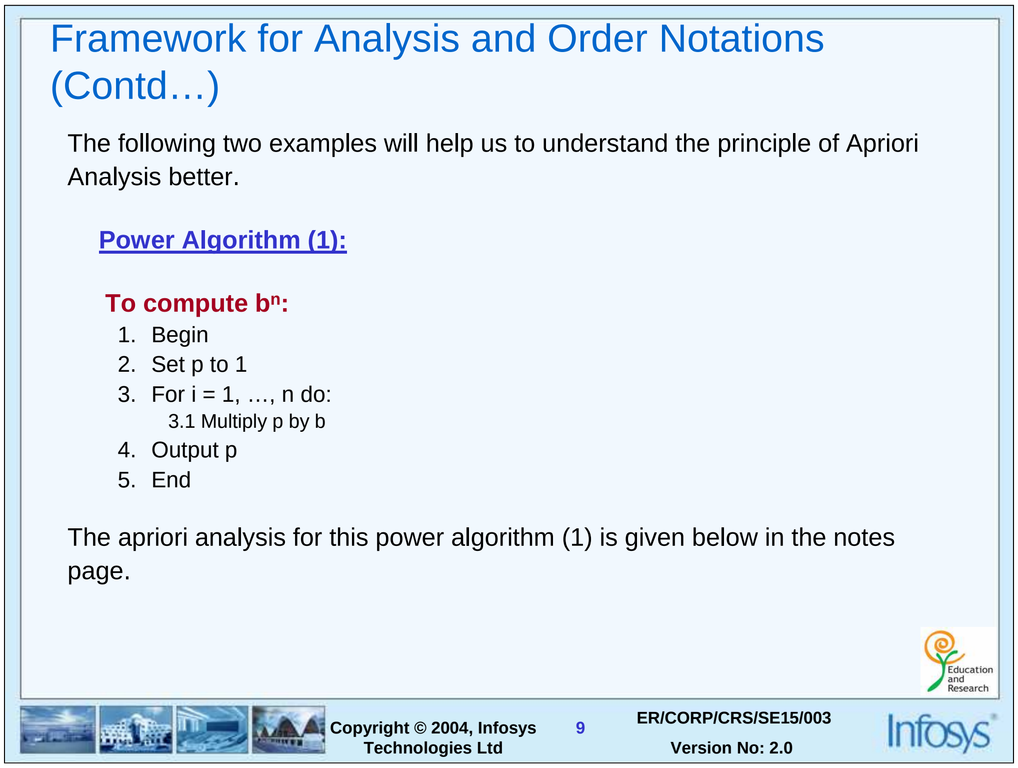
- ***Instruction Count***
  - ***Micro Analysis***
  - ***Macro Analysis***
- ***Worst case running time:*** The goodness of an algorithm is most often expressed in terms of its worst-case running time. There are two reasons for this: the need for a bound on one’s pessimism, and the ease of calculation of worst-case times as compared to average-case times
- ***Average Case running time:*** Average-case running times are calculated by first arriving at an understanding of the average nature of the input, and then performing a running-time analysis of the algorithm for this configuration.

•While doing an instruction count we can do either a **Micro Analysis** or a **Macro Analysis**

•Micro Analysis → Perform the instruction count for all operations

•Macro Analysis → Perform the instruction count only for dominant operations. The following are a few examples of dominant (or basic) operations.

- •Comparisons and Swapping are basic operations in sorting algorithms
- •Arithmetic operations are basic operations in math algorithms
- •Comparisons are basic operations in searching algorithms
- •Multiplication and Addition are basic operations in matrix multiplication algorithms



Framework for Analysis and Order Notations (Contd…)

The following two examples will help us to understand the principle of Apriori Analysis better.

Power Algorithm (1):

To compute b^n:

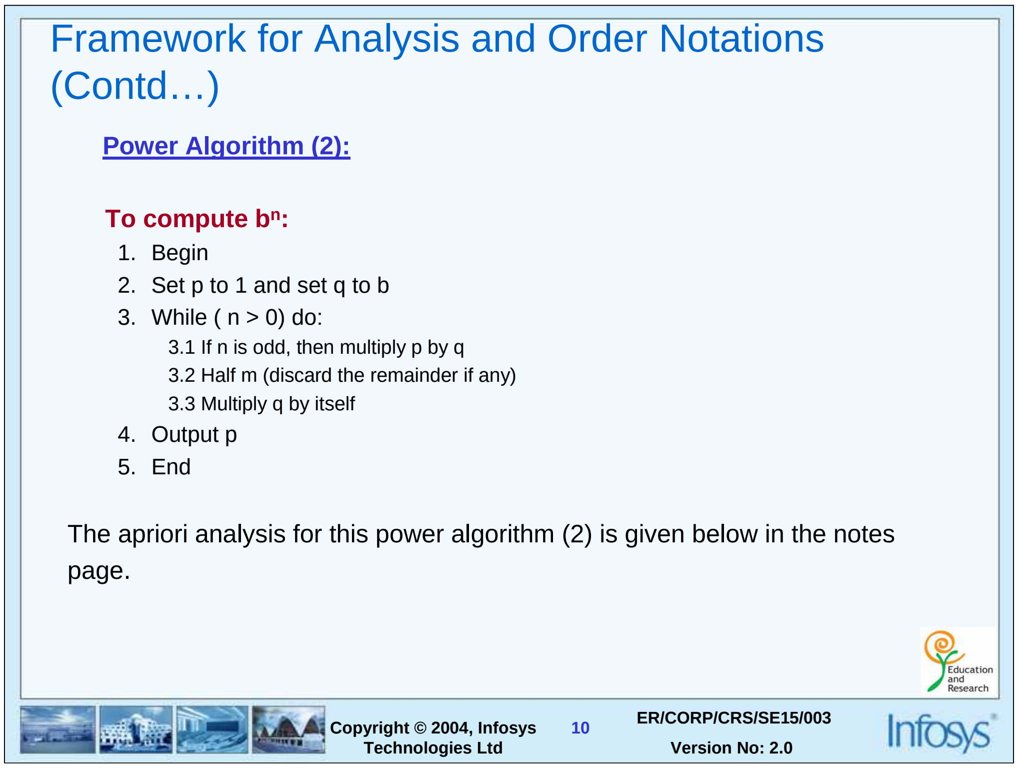
```
1. Begin
2. Set p to 1
3. For i = 1, …, n do:
      3.1 Multiply p by b
4. Output p
5. End
```

The apriori analysis for this power algorithm (1) is given below in the notes page.

The basic operation for this power algorithm is multiplication.

Macro Analysis: **Step 3.1** performs the multiplication operation and this step is performed **n** times. So the worst case complexity (or running time) for this power algorithm (1) is **n.**

Micro Analysis: **Step 2** is an assignment statement which is counted as **one unit**. **Step 3** is a for loop inside in which it performs the multiplication operation (**step 3.1**) which is again counted as **one unit** of operation. As **step 3** executes **n** times and each time it performs **one** multiplication, the total number of operations performed by **step 3** is **n**. **Step 4** is an output statement which is counted as **one unit**. So the worst case complexity for this power algorithm (1) is **(1 + n + 1) =(n+2)**.



Framework for Analysis and Order Notations (Contd…)

Power Algorithm (2):

To compute b^n:

```
1. Begin
2. Set p to 1 and set q to b
3. While ( n > 0) do:
      3.1 If n is odd, then multiply p by q
      3.2 Half m (discard the remainder if any)
      3.3 Multiply q by itself
4. Output p
5. End
```

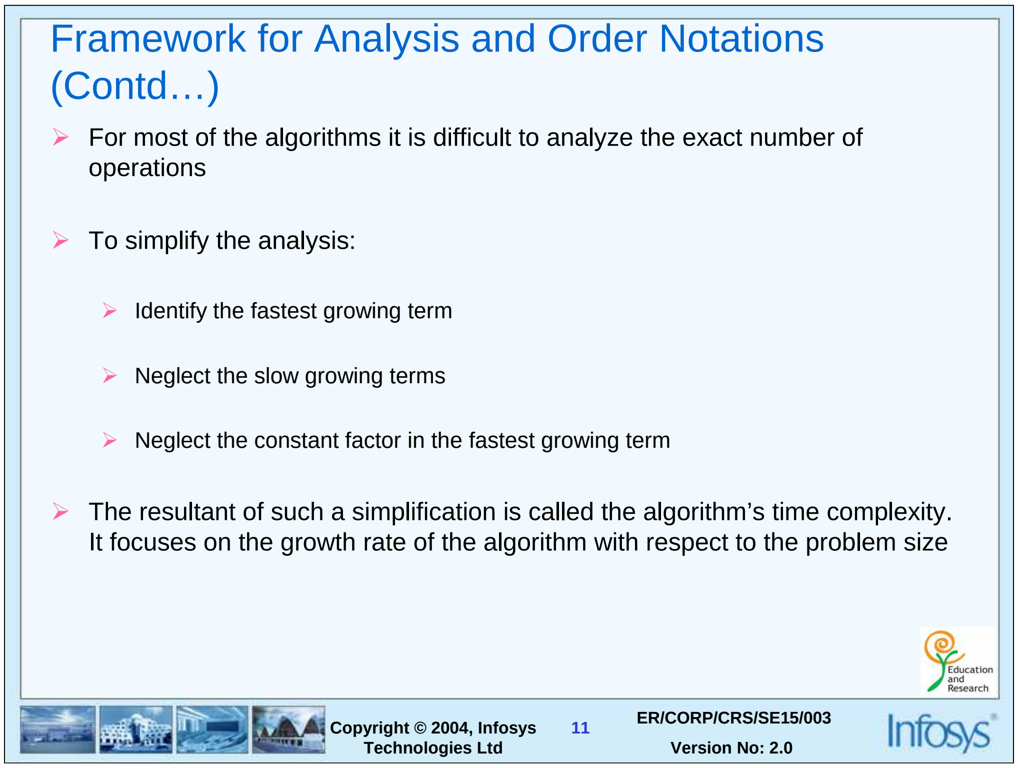
The apriori analysis for this power algorithm (2) is given below in the notes page.

The basic operation for this power algorithm is multiplication.

Macro Analysis: **Step 3.1** and **Step 3.3** perform the multiplication operation. Since we are interested in worst case analysis, we need to be as pessimistic as possible. So assuming that both these multiplication steps are executed every time the loop executes, we see that **2** multiplication operations are executed **k** times where **k** is the number of times the loop executes. So the worst case complexity (or running time) for this power algorithm (2) is **$2 * k = 2 * (\mathrm{floor}(\log_2(n)) + 1)$** . (We had seen in chapter – 1 that the number of times n must be divided by 2 to reach zero is (floor(log(n)) + 1))

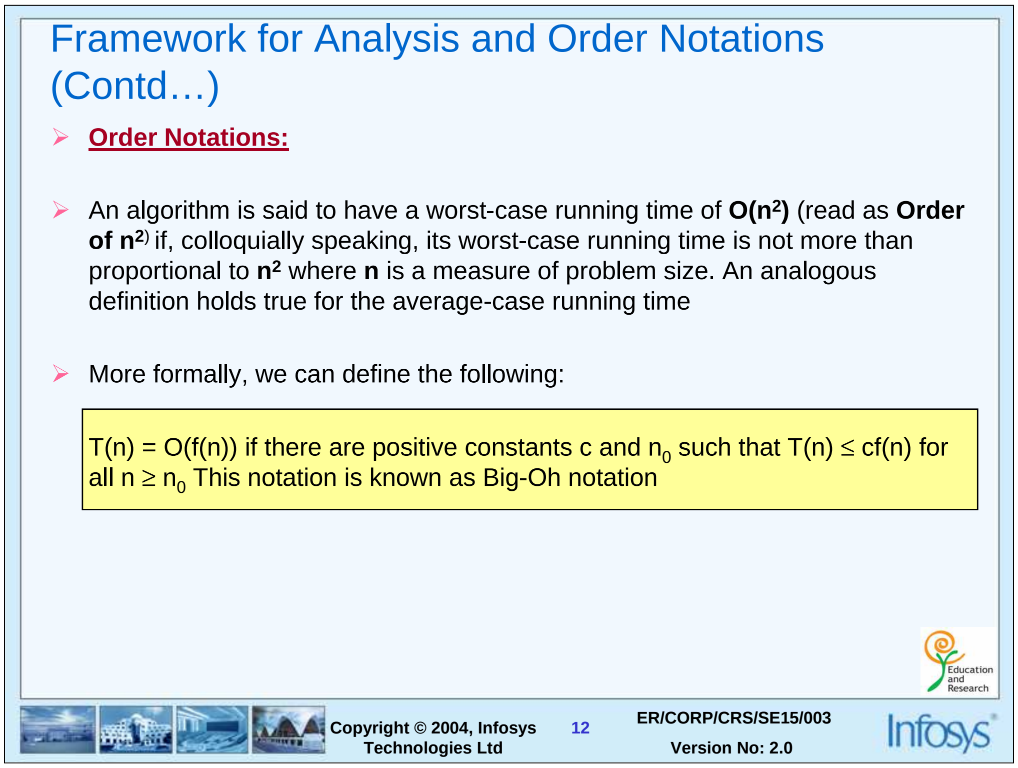
Micro Analysis: **Step 2** is an assignment statement which is counted as **one unit**. **Step 3** is a while loop inside in which it performs the multiplication operation (**step 3.1 and 3.3**) and a division operation (**step 3.2**). Each multiplication operation and division operation is counted as **one unit** of operation. As **step 3** executes **$(\mathrm{floor}(\log_2(n)) + 1)$** times and each time it performs **two** multiplications (in the worst case) and **one** division, the total number of operations performed by **step 3** is **$(3 * (\mathrm{floor}(\log_2(n)) + 1))$**. **Step 4** is an output statement which is counted as **one unit**. So the worst case complexity for this power algorithm (2) is

$(1 + (3 * (\mathrm{floor}(\log_2(n)) + 1)\ ) + 1) =((3 * (\mathrm{floor}(\log_2(n))) +5)$


Framework for Analysis and Order Notations (Contd…)

- For most of the algorithms it is difficult to analyze the exact number of operations
- To simplify the analysis:
  - Identify the fastest growing term
  - Neglect the slow growing terms
  - Neglect the constant factor in the fastest growing term
- The resultant of such a simplification is called the algorithm's time complexity. It focuses on the growth rate of the algorithm with respect to the problem size

The need for such a simplification arises as it is practically difficult to do a precise mathematical analysis for most of the algorithms. Such a simplification (as mentioned above) leads to the concept of growth rate. Once we are interested in growth rate and not the exact number of operations, it makes sense (both mathematically and logically) to ignore the slow growing terms and also the co-efficient of the fastest growing term. The Order Notations which we will study shortly are related to the growth rate.



Framework for Analysis and Order Notations (Contd…)

- **Order Notations:**

- An algorithm is said to have a worst-case running time of **O(n^2)** (read as **Order of n^2)** if, colloquially speaking, its worst-case running time is not more than proportional to **n^2** where **n** is a measure of problem size. An analogous definition holds true for the average-case running time

- More formally, we can define the following:

T(n) = O(f(n)) if there are positive constants c and n_0 such that T(n) $\leq$ cf(n) for all n $\geq$ n_0 This notation is known as Big-Oh notation

12 ER/CORP/CRS/SE15/003 Version No: 2.0

The Big-Oh notation is more like an upper bound. Since it is tough to find the exact number of operations performed by an algorithm in terms of its worst case complexity, we are looking at proportional growth rates. The Big-Oh notation can be viewed as an upper bound for the worst case complexity.

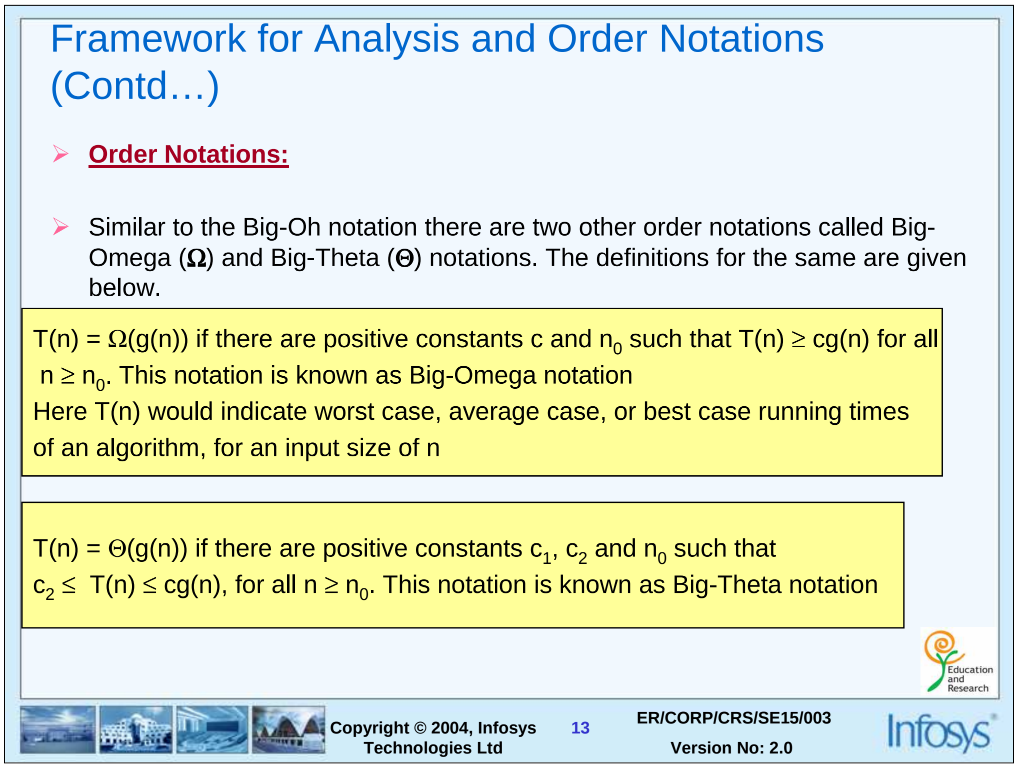
Suppose that we have a set of numbers ranging from a to b. Then any number greater than or equal to b (it could be b or b+1, or b+2, …) is an upper bound for this set. As the number increases from a to b it moves closer to this upper bound but at no point of time it is greater than this upper bound.

When we try to extend a similar kind of principle to functions, we see that up to a certain problem size this relationship with the upper bound may not hold true. Beyond that particular problem size all possible inputs for the function falls in line with the growth rate of the upper bound. This is precisely the Big-Oh notation.

Informally O(f(n)) denotes the set of all functions with a smaller or same order of growth as f(n). For example, n^2 belongs O(n^3).

When we apply the simplification procedure to the power algorithm (2) we get the following:

2 * k = 2 * floor($\log_2$(n)) + 2 which is **2 * floor($\log_2$(n))** (ignoring the slow growing terms) Which is **floor($\log_2$(n))** (neglecting the coefficient of the fast growing term) which is nothing but **O(log(n))**. So the worst case complexity of the power algorithm (2) is **O(log(n))**.



Framework for Analysis and Order Notations (Contd…)

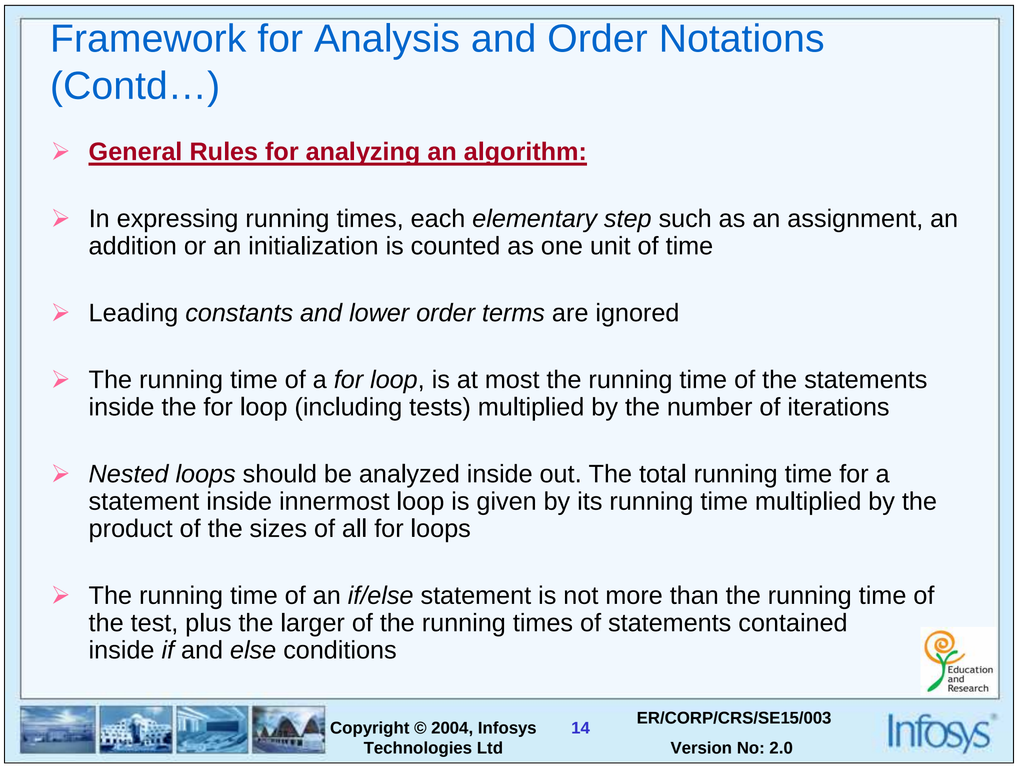
- **Order Notations:**
- Similar to the Big-Oh notation there are two other order notations called Big-Omega (**Ω**) and Big-Theta (**Θ**) notations. The definitions for the same are given below.

$T(n) = \Omega(g(n))$ if there are positive constants c and n_0 such that $T(n) \geq cg(n)$ for all $n \geq n_0$. This notation is known as Big-Omega notation
Here T(n) would indicate worst case, average case, or best case running times of an algorithm, for an input size of n

$T(n) = \Theta(g(n))$ if there are positive constants c_1, c_2 and n_0 such that $c_2 \leq T(n) \leq cg(n)$, for all $n \geq n_0$. This notation is known as Big-Theta notation

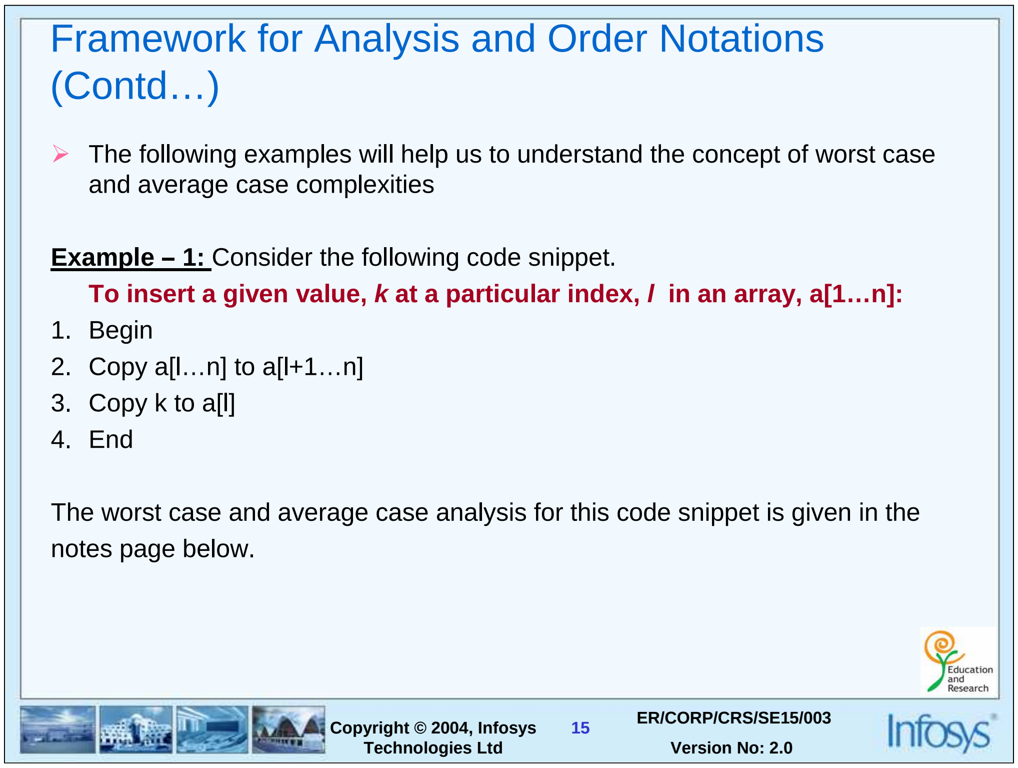
The Big-Omega notation can be considered as a lower bound for the T(n) which is the actual running time of an algorithm. Informally Ω(g(n)) denotes the set of all functions with a larger or same order of growth as g(n). For example, n^2 belongs Ω(n).

Informally the Θ(g(n)) is the set of all functions which have the same order of growth as g(n). For example, $n^2 + 100n + 500 = \Theta(n^2)$. We are normally not interested in the Big-Omega and Big-Theta notations. When ever we refer to the complexity of an algorithm we mean the worst case complexity which is expressed in terms of the Big-Oh notation. In other words when we say that an algorithm is of order of f(n) we basically mean that the worst case complexity of the algorithm is f(n).



Framework for Analysis and Order Notations (Contd…)

- **General Rules for analyzing an algorithm:**
- In expressing running times, each *elementary step* such as an assignment, an addition or an initialization is counted as one unit of time
- Leading *constants and lower order terms* are ignored
- The running time of a *for loop*, is at most the running time of the statements inside the for loop (including tests) multiplied by the number of iterations
- *Nested loops* should be analyzed inside out. The total running time for a statement inside innermost loop is given by its running time multiplied by the product of the sizes of all for loops
- The running time of an *if/else* statement is not more than the running time of the test, plus the larger of the running times of statements contained inside *if* and *else* conditions



Framework for Analysis and Order Notations (Contd…)

- The following examples will help us to understand the concept of worst case and average case complexities

<u>Example – 1:</u> Consider the following code snippet.

To insert a given value, *k* at a particular index, *l* in an array, a[1…n]:

```
1. Begin
2. Copy a[l…n] to a[l+1…n]
3. Copy k to a[l]
4. End
```

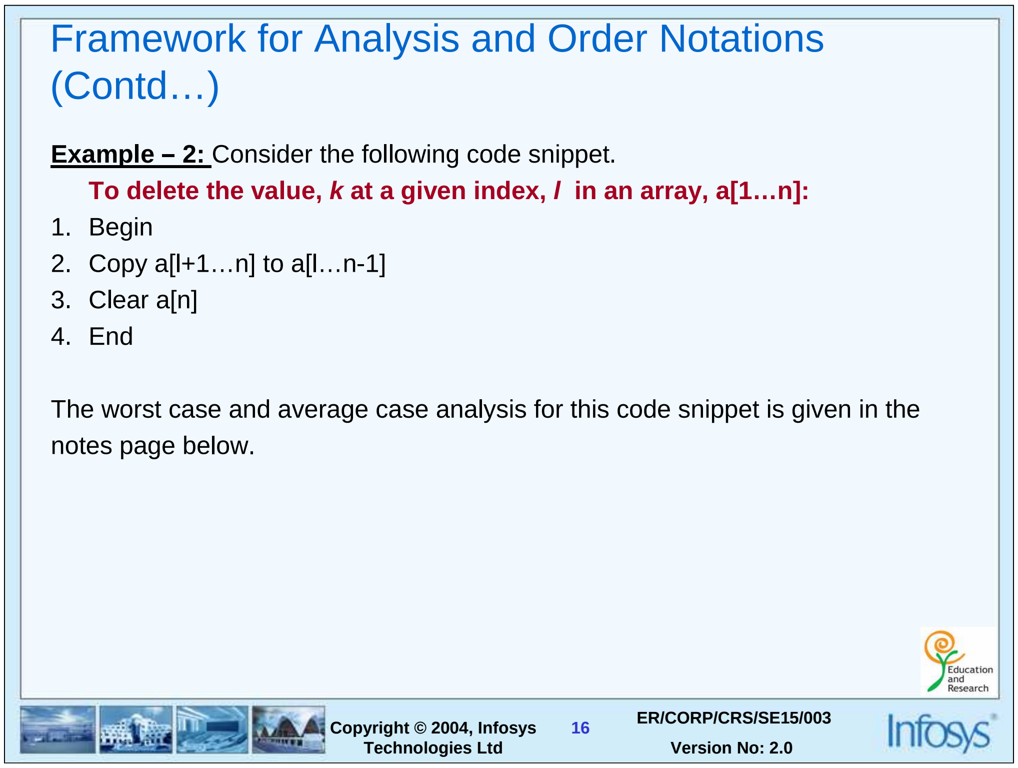
The worst case and average case analysis for this code snippet is given in the notes page below.

ER/CORP/CRS/SE15/003
Version No: 2.0

The above given code inserts a value **k** into position **l** in an array **a**. The basic operation here is **copy**.

<u>Worst Case Analysis:</u> Step 2 does **n-1** copies in the worst case. **Step 3** does **1** copy. So the total number of copy operations is **n-1+1=n**. Hence the worst case complexity of array insertion is **O(n)**.

<u>Average Case Analysis:</u> On an average **step 2** will perform **(n-1)/2** copies. This is derived as follows: The probability that **step 2** performs **1** copy is **1/n**, the probability that it performs **2** copies is **2/n** and so on. The probability that it performs **n-1** copies is **(n-1)/n**. Hence the average number of copies that **step 2** performs is **(1/n) + (2/n) + … + (n-1)/n = (n-1)/2**. Also **step 3** performs **1** copy. So on an average the array insertion performs **((n-1)/2) + 1** copies. Hence the average case complexity of array insertion is **O(n)**.



Framework for Analysis and Order Notations (Contd…)

Example – 2: Consider the following code snippet.

To delete the value, *k* at a given index, *l* in an array, a[1…n]:

```
1. Begin
2. Copy a[l+1…n] to a[l…n-1]
3. Clear a[n]
4. End
```

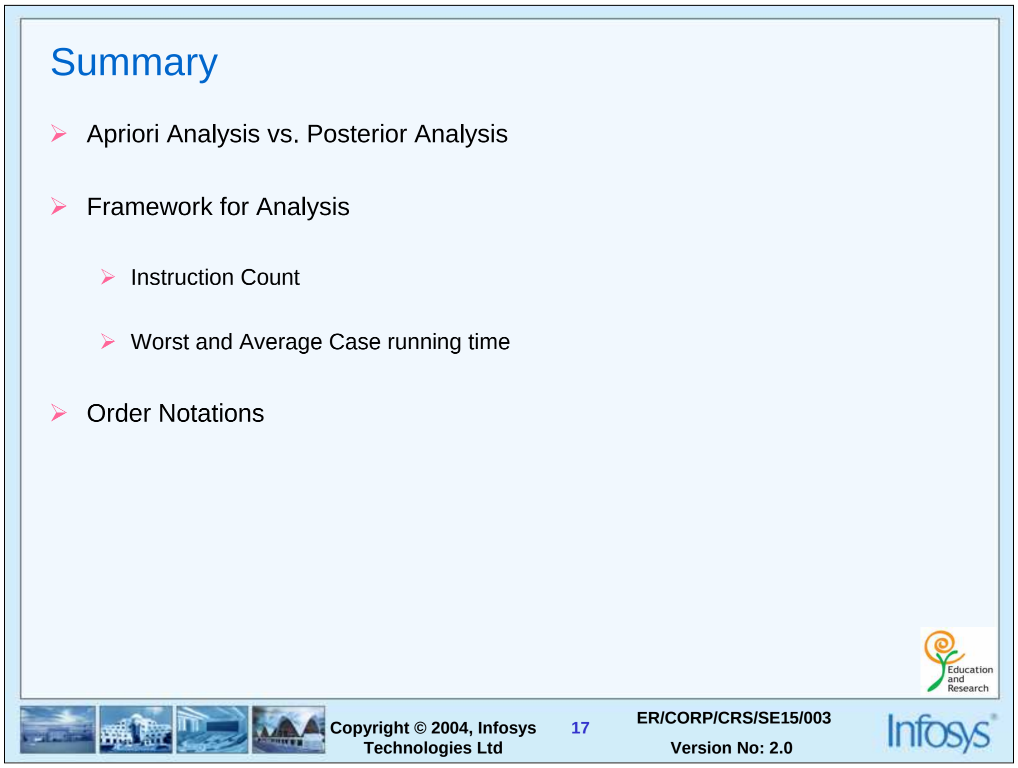
The worst case and average case analysis for this code snippet is given in the notes page below.

The above given code deletes the value **k** at a given index **l** in an array **a**. The basic operation here is **copy**.

Worst Case Analysis: **Step 2** does **n-1** copies in the worst case. So the total number of copy operations is **n-1**. Hence the worst case complexity of array insertion is **O(n)**.

Average Case Analysis: On an average **step 2** will perform **(n-1)/2** copies. This is derived as follows: The probability that **step 2** performs **1** copy is **1/n**, the probability that it performs **2** copies is **2/n** and so on. The probability that it performs **n-1** copies is **(n-1)/n**. Hence the average number of copies that **step 2** performs is **(1/n) + (2/n) + … + (n-1)/n = (n-1)/2**. So on an average the array deletion performs

((n-1)/2) copies. Hence the average case complexity of array insertion is **O(n)**.



Summary

- Apriori Analysis vs. Posterior Analysis
- Framework for Analysis
  - Instruction Count
  - Worst and Average Case running time
- Order Notations

In this chapter we had:

•Understood the benefits of doing an apriori analysis when compared to a posterior analysis

•Studied the framework for analyzing algorithms by doing an instruction count on the input size and deriving the worst case running time

•Studied the different order notations used for analyzing algorithms and understood, with the aid of a few examples, the art of doing worst case and average case analysis for a given algorithm