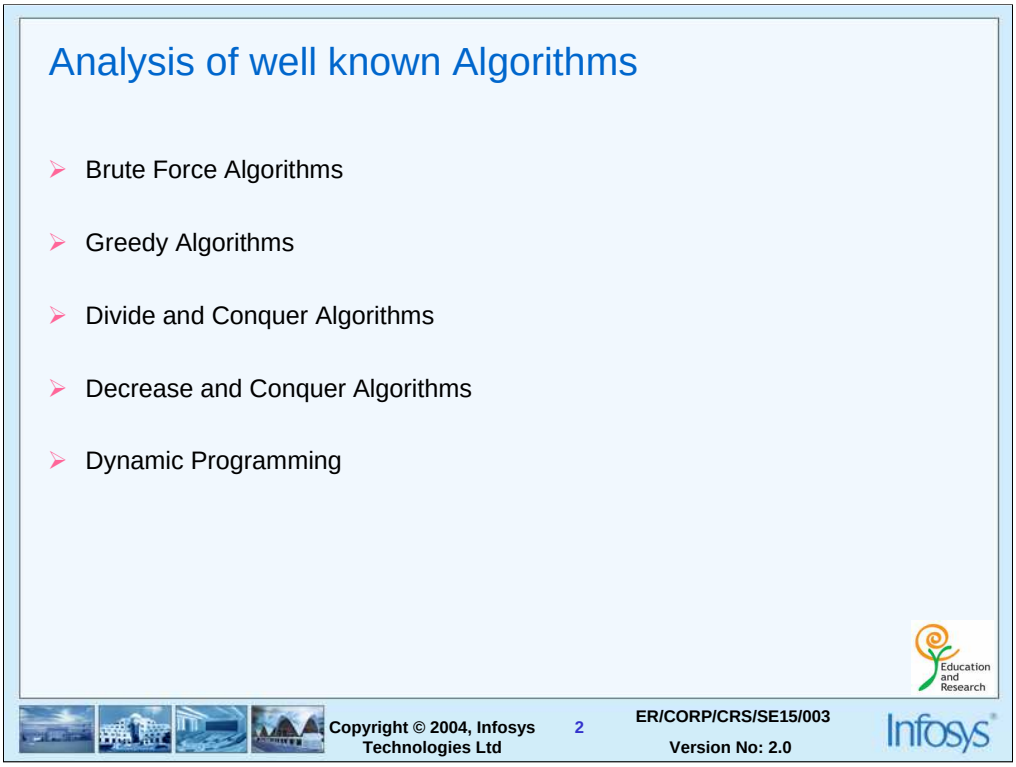
Infosys®
POWERED BY INTELLECT
DRIVEN BY VALUES

Analysis of Algorithms

Unit 4 - Analysis of well known Algorithms

Education and Research



Analysis of well known Algorithms:

- In this chapter we will understand the design methodologies of certain well algorithms and also analyze the complexities of these well known algorithms.

Brute Force Algorithms

- Brute force approach is a straight forward approach to solve the problem. It is directly based on the problem statement and the concepts
- The force here refers to the computer and not to the intellect of the problem solver
- It is one of the simplest algorithm design to implement and covers a wide range of problems under its gamut

We will analyze the following three algorithms which fall under the category of brute force algorithm design.

- Selection Sort
- Bubble Sort
- Linear Search

Brute Force Algorithms – Selection Sort

- Sorting is the technique of rearranging a given array of data into a specific order. When the array contains numeric data the sorting order is either ascending or descending. Similarly when the array contains non-numeric data the sorting order is lexicographical order
- All sorting algorithms start with the given input array of data (unsorted array) and after each iteration extend the sorted part of the array by one cell. The sorting algorithm terminates when sorted part of array is equal to the size of the array
- In selection sort, the basic idea is to find the smallest number in the array and swap this number with the leftmost cell of the unsorted array, thereby increasing the sorted part of the array by one more cell

In all sorting algorithms the entire array is seen as two parts, the sorted and unsorted part. Initially the size of the sorted part is zero and the unsorted part is the entire array. The basic aim of sorting is to extend the sorted part of the array after each iteration of the algorithm thereby decreasing the unsorted part. The algorithm terminates when the sorted part is the entire array (or in other words the unsorted part becomes zero)

Brute Force Algorithms – Selection Sort (Contd…)

Selection Sort:

To sort the given array a[1…n] in ascending order:

1. Begin

2. For i = 1 to n-1 do
   - 2.1 set min = i
   - 2.2 For j = i+1 to n do
     - 2.2.1 If (a[j] < a[min]) then set min = j
   - 2.3 If (i ≠ min) then swap a[i] and a[min]

3. End

Let us analyze the selection sort. The basic operation here is comparison and swapping.

Since we have a nested for loop we need to do an inside out analysis. **Step 2.2.1** does one comparison and this step executes **(n - (i+1) +1)** times. **Step 2.3** performs **1** swap operation (assuming that each time the if condition is true). So number of basic operations performed inside the first for loop (**step 2)** is

(**(n – (i+1) +1) +1) = (n-i+1)**. From **step 2** we can see that **i** varies from **1** to **n-1**. So the number of basic operations performed by the outer for loop is:

(n-1+1) + (n-2+1) + (n-3+1) + … + (n-(n-1)+1) = n + (n-1) + (n-2) + 2

= ((n(n+1))/2 – 1) = (n^2 + n – 2)/2 . Ignoring the slow growing terms we get the worst case complexity of selection sort as **O(n^2)**

Brute Force Algorithms – Bubble Sort

- Bubble sort is yet another sorting technique which works on the design of brute force
- Bubble sort works by comparing adjacent elements of an array and exchanges them if they are not in order
- After each iteration (pass) the largest element bubbles up to the last position of the array. In the next iteration the second largest element bubbles up to the second last position and so on
- Bubble sort is one of the simplest sorting techniques to implement

Brute Force Algorithms – Bubble Sort (Contd…)

Bubble Sort:

To sort the given array a[1…n] in ascending order:

1. Begin

2. For i = 1 to n-1 do
   2.1 For j = 1 to n-1-i do
      2.2.1 If (a[j+1] < a[j]) then swap a[j] and a[j+1]

3. End

The analysis of bubble sort is given below. The basic operation here again is comparison and swapping

We have a nested for loop and hence we need to do an inside out analysis. **Step 2.2.1** at the most performs **2** operations (one comparison and one swapping) and these operations are repeated **((n-1-i) – 1) + 1** times. i.e. **Step 2.1** performs **(n-i-1)** operation where **i** varies from **1** to **n-1 (step 2**). So the total number of operations performed by bubble sort is :

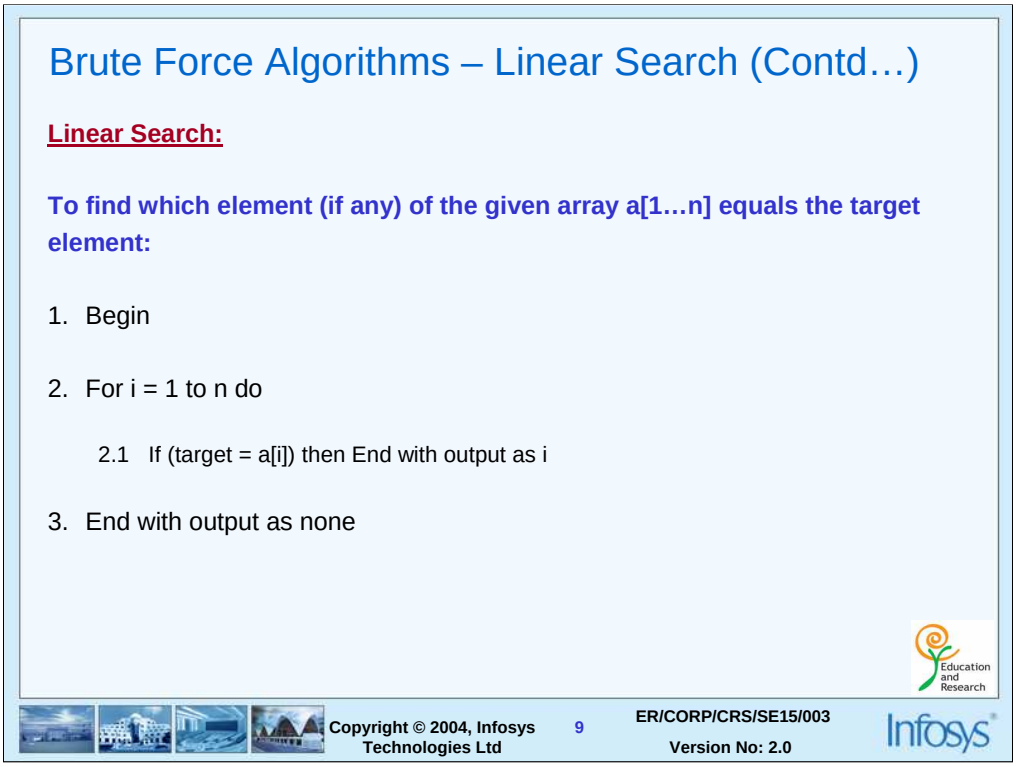
(n-1-1)+(n-2-1)+(n-3-1)+…+(n-(n-1) – 1) = (n-2)+(n-3)+(n-4)+…+2

= ((n-2)(n-1))/2 – 1 = (n^2-3n)/2.

Hence the worst case complexity of bubble sort is **O(n^2)**

Brute Force Algorithms – Linear Search

- Searching is a technique where in we find if a target element is part of the given set of data
- There are different searching techniques like linear search, binary search etc…
- Linear search works on the design principle of brute force and it is one of the simplest searching algorithm as well
- Linear search is also called as a sequential search as it compares the successive elements of the given set with the search key

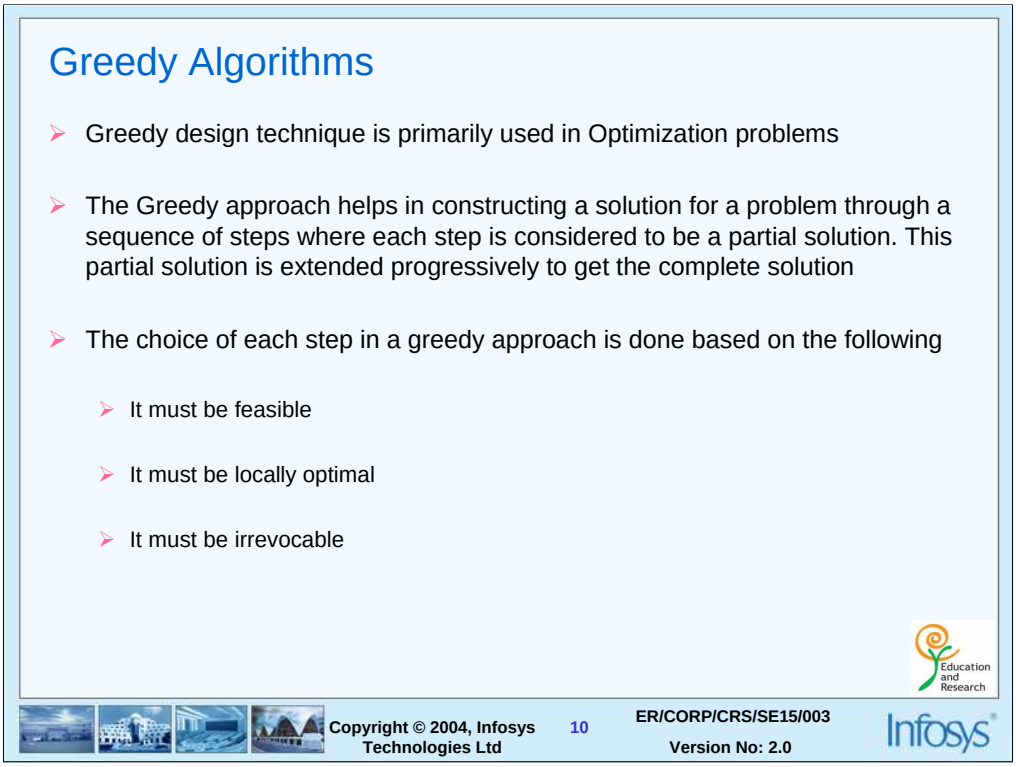


The analysis of Linear search is given below. The basic operation here is comparison.

Worst Case Analysis: Step 2.1 performs **1** comparison and this step executes **n** times. So the worst case complexity of Linear search is **O(n).**

Average Case Analysis: To perform an average case analysis we need to work with the assumption that every element in the given array is equal likely to be the target. So the probability that the search element is at position **1** is **1/n**, the probability that the search element is at position **2** is **2/n**, and so on. The probability that the search element is at the last position is **n/n**. Hence the average case complexity of the linear search is:

(1/n)+(2/n)+…+(n/n) = (1/n)(1+2+…+n) = (1/n)(n(n+1)/2) = (n+1)/2 → O(n)



Optimization problems are problems where in we would like to find the best of all possible solutions. In other words we need to find the solution which has the optimal (maximum or minimum) value satisfying the given constraints.

In the greedy approach each step chosen has to satisfy the constraints given in the problem. Each step is chosen such that it is the best alternative among all feasible choices that are available. The choice of a step once made cannot be changed in subsequent steps.

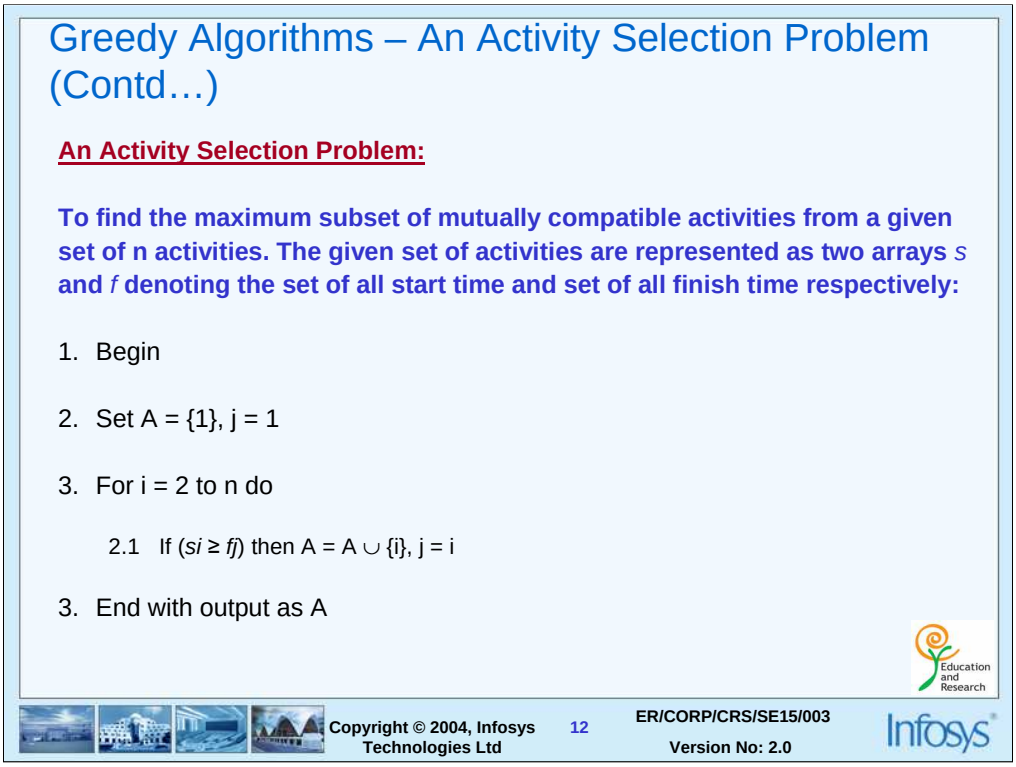
We analyze an Activity – Selection Problem which works on the greedy approach.

Greedy Algorithms – An Activity Selection Problem

- An Activity Selection problem is a slight variant of the problem of scheduling a resource among several competing activities.

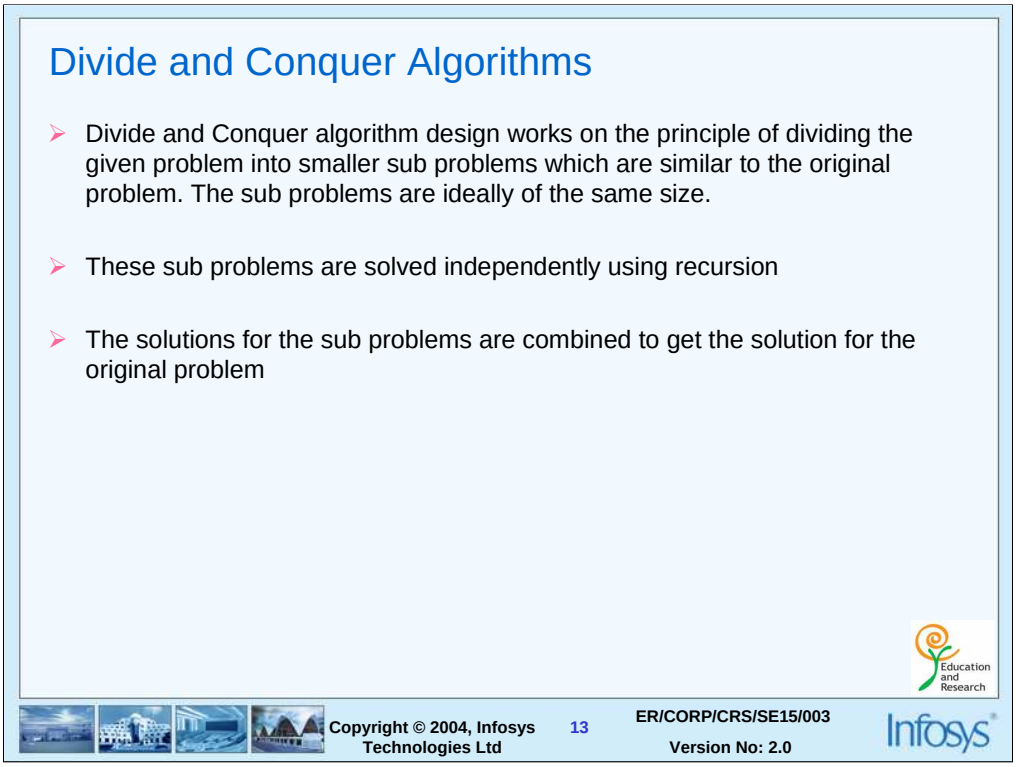
Suppose that we have a set **S = {1, 2, …, n}** of **n** events that wish to use an **auditorium** which can be used by only one event at a time. Each event **i** has a start time s_i and a finish time f_i where $s_i \leq f_i$. An event **i** if selected can be executed anytime on or after s_i and must necessarily end before f_i. Two events **i** and **j** are said to **compatible** if they do not overlap (meaning $s_i \geq f_j$ or $s_j \geq f_i$). The activity selection problem is to select a maximum subset of compatible activities.

To solve this problem we use the greedy approach



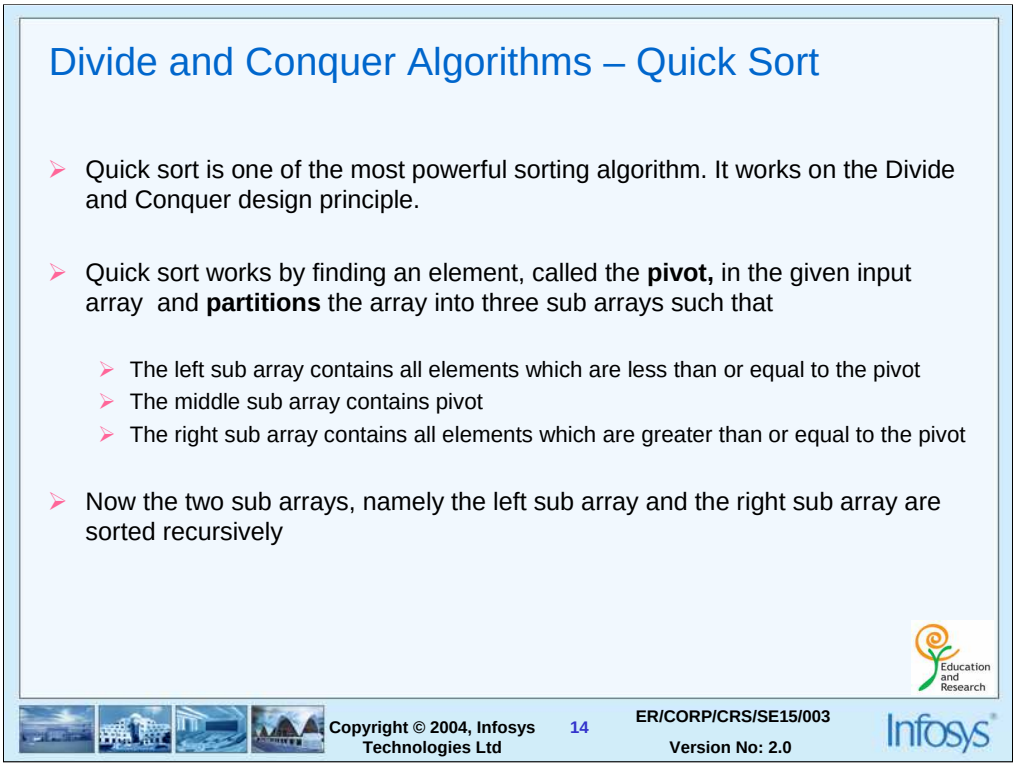
The Activity Selection Problem works under the assumption that the given input array *f* is sorted in the increasing order of the finishing time. The greedy approach in the above algorithm is highlighted in **step 2.1** when we select the next activity such that the start time of that activity is greater than or equal to the finish time of the preceding activity in the array **A**. Note that it suffices to do only this check. We don't have to compare the start time of the next activity with all the elements of the array **A**. This is so because the finishing time (array *f*) are in increasing order. The analysis of the activity-selector problem is given below.

Worst Case Analysis: The basic operation here is comparison. **Step 2.1** performs **1** comparison and this step is repeated **(n-1)** times. So the worst case complexity of this algorithm is **1 * (n-1) = O(n)**.

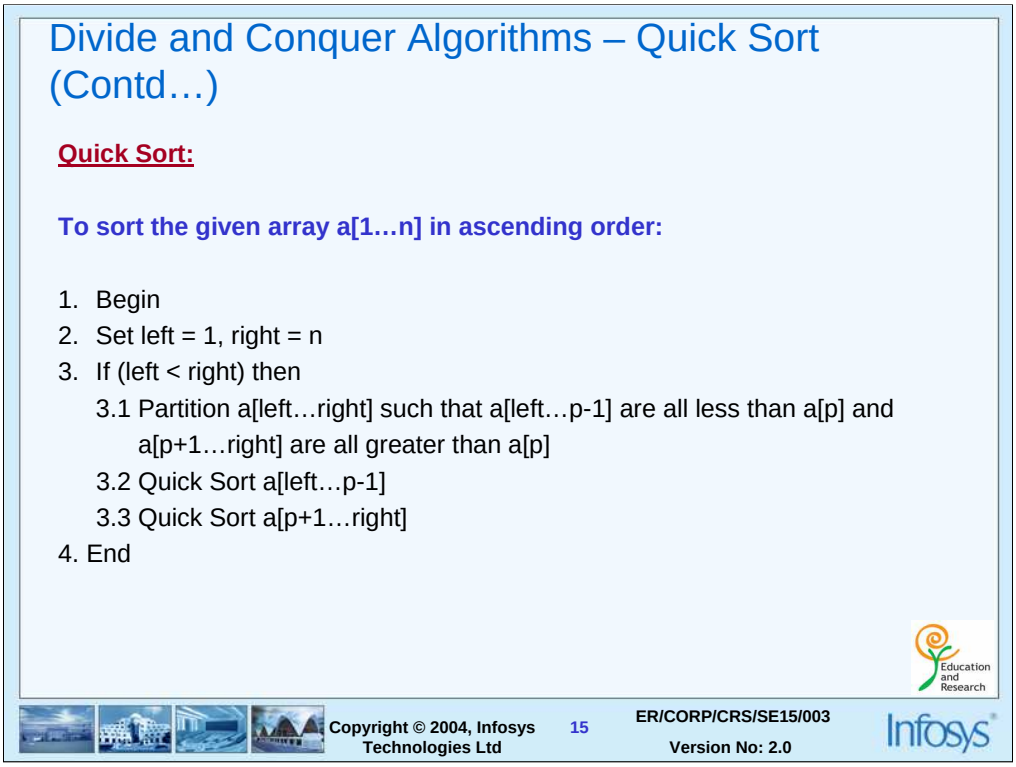


The Divide and Conquer strategy can be viewed as one which has three steps. The first step is called **Divide** which is nothing but dividing the given problems into smaller sub problems which are identical to the original problem and also these sub problems are of the same size. The second step is called **Conquer** where in we solve these sub problems recursively. The third step is called **Combine** where in we combine the solutions of the sub problems to get the solution for the original problem.

As part of the Divide and Conquer algorithm design we will analyze the Quick sort, Merge sort and the Binary Search algorithms.



The basic principle of Quick sort is given above. The **partitioning** of the given input array is part of the **Divide** step. The recursive calls to sort the sub arrays are part of the **Conquer** step. Since the sorted sub arrays are already in the right place there is no **Combine** step for the Quick sort.



The basic operation in Quick sort is comparison and swapping. Note that the Quick Sort Algorithm calls it self recursively. Quick Sort is one of the classical examples for recursive algorithms. There are many partition algorithms which identifies the pivot and partitions the given array. We will see one simple partition algorithm which takes **n** comparisons to partition the given array.

Analysis of Quick Sort: **Step 3.1** partitions the given array. The partition algorithm which we will see shortly performs **(n-1)** comparisons to identify the pivot and rearrange the array such that every element to the left of the pivot is smaller than or equal to the pivot and every element to the right of the pivot is greater than or equal to the pivot. Hence **Step 3.1** performs **(n-1)** operations. **Step 3.2** and **Step 3.3** are recursive calls to Quick Sort with a smaller problem size.

Divide and Conquer Algorithms – Quick Sort (Contd…)

The Best Case and Worst Case Analysis for Quick Sort is given below in the notes page

To analyze recursive algorithms we need to solve the recurrence relation. Let **T(n)** denote the total number of basic operations performed by Quick Sort to sort an array on **n** elements.

Best Case Analysis: In **best case** the the partition algorithm will split the given array into **2 equal** sub arrays. i.e. In the **Best Case**, the **Pivot** turns out to be the **median value** in the array. Then both the left sub array and the right sub array are both of equal size (**n/2**). Hence

$T(n) = 2T(n/2) + (n-1)$, if $n \geq 1$

$= 0$, if $n \leq 1$

$\Rightarrow T(n) = 2(2T(n/4) + ((n/2)-1)) + n-1 = 2^2\, T(n/2^2) + 2n - 3$, simplifying this recurrence in a similar manner we get,

$T(n) = 2^k\, T(n/2^k) + kn - (2^k - 1)$. The base condition for the recurrence is **T(1)**. So in this case we have **$n/2^k = 1 \Rightarrow k = \log(n)$**. Substituting for k we get

$T(n) = n + n\log(n) - (n-1) \Rightarrow T(n) = n\log n + 1$.

$\Rightarrow$ The **Best Case Complexity** of quick sort if **O(nlogn)**.

Worst Case Analysis: The worst case for Quick sort can arise when the partition algorithm divides the array in such a way that either the left sub array or the right sub array is empty. In which case the problem size is reduced from **n** to **n-1**. The recurrence relation for the worst case is as follows:

$T(n) = T(n-1) + n-1$, if $n > 1$

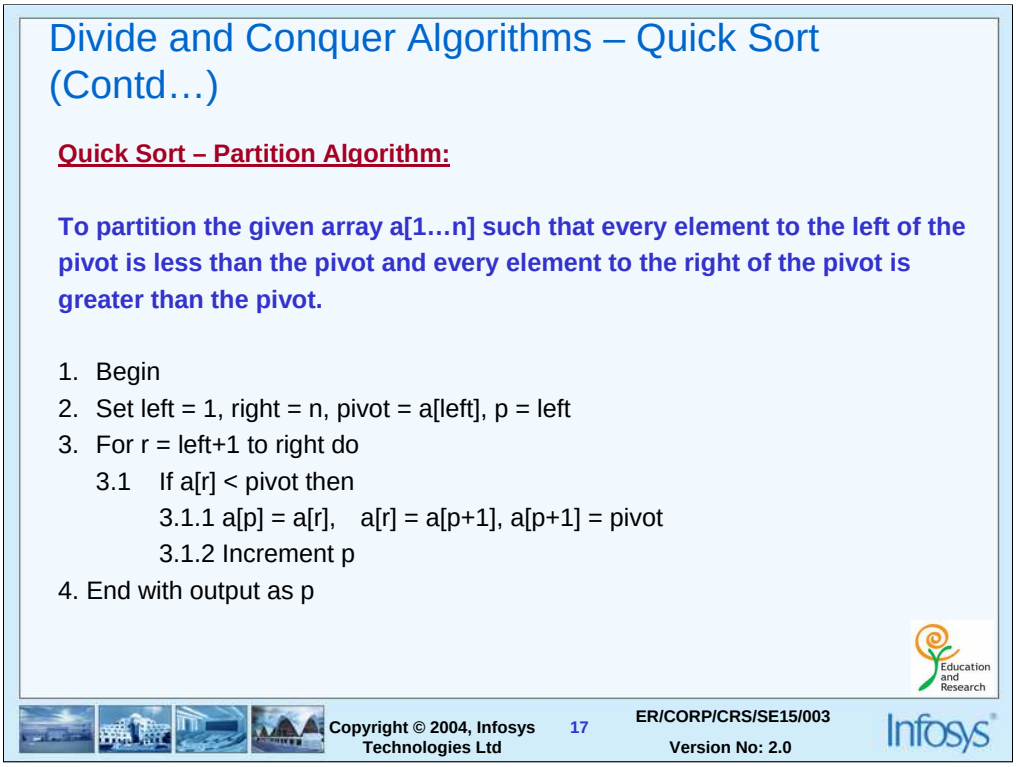
$= 0$, if $n \leq 1$

Simplifying this we get, **$T(n) = (n-1) + (n-2) + \ldots + 2 + 1 = ((n-1)*n)/2$**

$\Rightarrow$ The **Worst Case Complexity** of Quick Sort is **$O(n^2)$**.

The **Average Case Complexity** of Quick Sort is **O(nlogn)**

Remark: The Worst Case for the Quick Sort could happen when the pivot element is the smallest (or largest) in each recursive call.



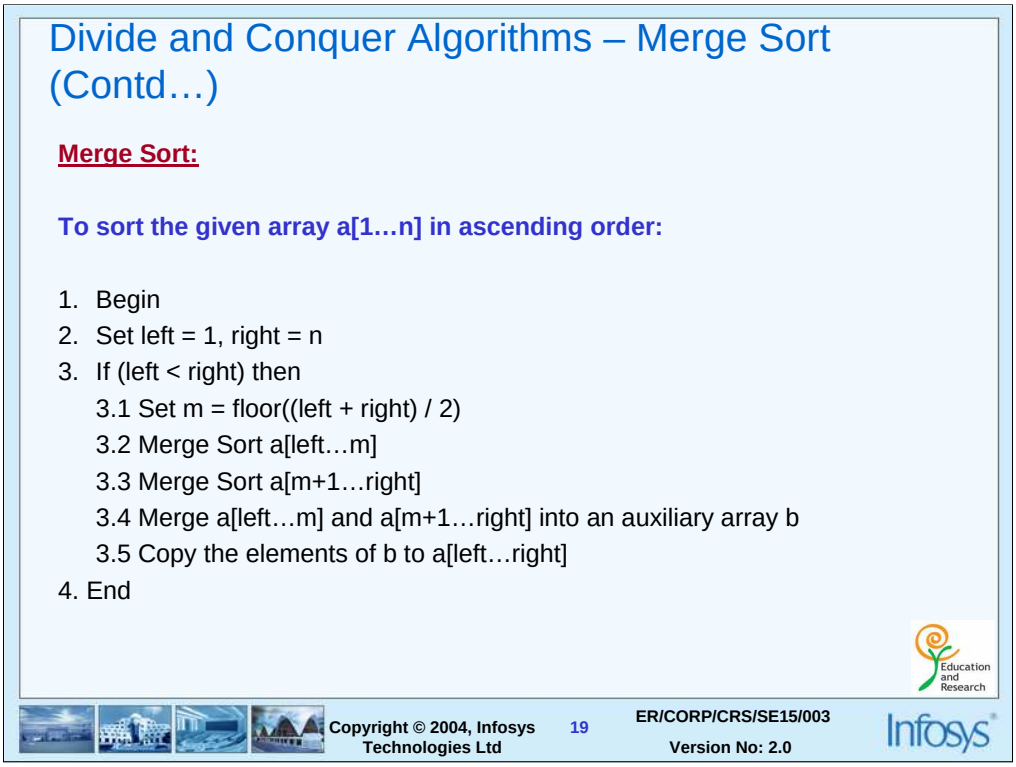
The partition algorithm given is one of the simplest partition algorithms for quick sort. The basic operation here is *comparison*.

Worst Case Analysis: **Step 3.1** performs **1** comparison and this step is repeated **n-1** times. So the **Worst Case Complexity** of the above mentioned partition algorithm is **O(n)**.

Divide and Conquer Algorithms – Merge Sort

- Merge sort is yet another sorting algorithm which works on the Divide and Conquer design principle.
- Merge sort works by dividing the given array into two sub arrays of equal size
- The sub arrays are sorted independently using recursion
- The sorted sub arrays are then merged to get the solution for the original array

The **breaking** of the given input array into two sub arrays of equal size is part of the **Divide** step. The recursive calls to sort the sub arrays are part of the **Conquer** step. The merging of the sub arrays to get the solution for the original array is part of the **Combine** step.



Again the basic operation in Merge sort is comparison and swapping. Merge Sort Algorithm calls it self recursively. Merge Sort divides the array into sub arrays based on the *position* of the elements whereas Quick Sort divides the array into sub arrays based on the value of the elements. Merge Sort requires an auxiliary array to do the *merging* (**Combine step**). The *merging* of two sub arrays, which are already sorted, into an auxiliary array can be done in **O(n)** where **n** is the total number of elements in both the sub arrays. This is possible because both the sub arrays are sorted.

Worst Case Analysis of Merge Sort: **Step 3.2** and **Step 3.3** are recursive calls to Merge Sort with a smaller problem size. Also **step 3.4** is the *merging* step which takes **O(n)** comparisons. We need to first identify the recurrence relation for this algorithm. Let **T(n)** denote the total number of basic operations performed by Merge Sort to sort an array on **n** elements.

T(n) = 2 T(n/2) + n, if n > 1

= 0, if n ≤1

This when simplified gives **T(n) = O(nlogn)**. Hence the **Worst Case Complexity** of Merge Sort is **O(nlogn)**.

Divide and Conquer Algorithms – Binary Search

- Binary Search works on the Divide and Conquer Strategy
- Binary Search takes a sorted array as the input
- It works by comparing the target (search key) with the middle element of the array and terminates if it is the same, else it divides the array into two sub arrays and continues the search in left (right) sub array if the target is less (greater) than the middle element of the array.

Reducing the problem size by half is part of the **Divide** step in Binary Search and searching in this reduced problem space is part of the **Conquer** step in Binary Search. There is no **Combine** step in Binary Search.

Divide and Conquer Algorithms – Binary Search (Contd…)

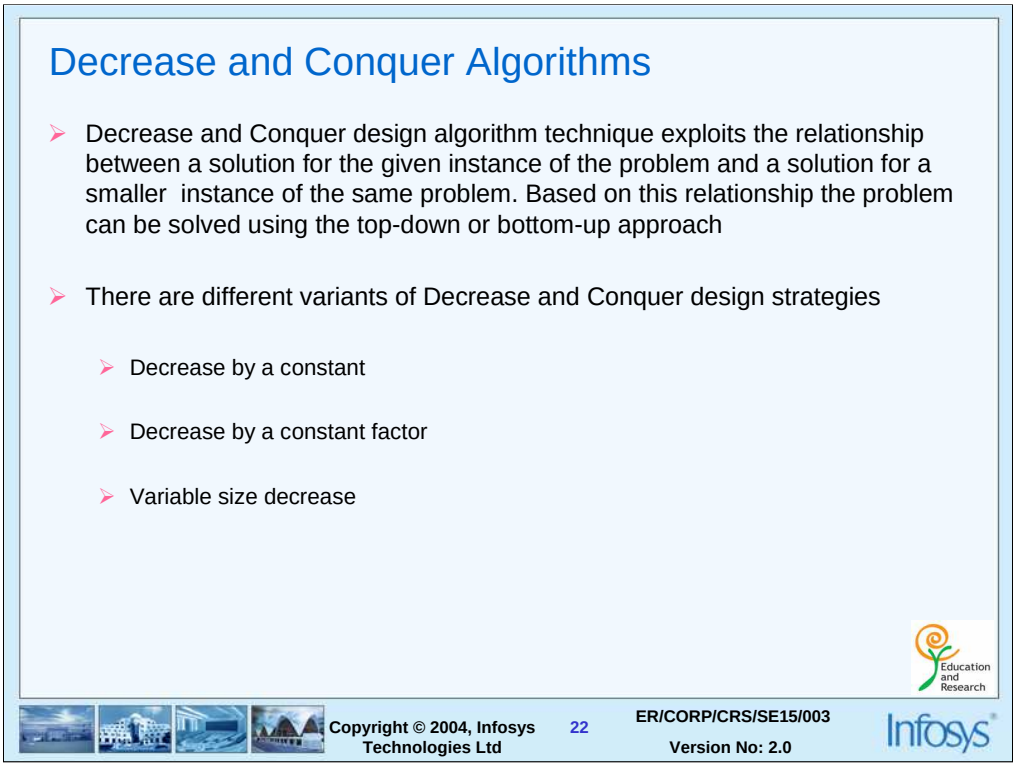
Binary Search:

To find which element (if any) of the given array a[1…n] equals the target element:

1. Begin
2. Set left = 1, right = n
3. While (left ≤ right) do
   3.1 m = floor((left + right)/2)
   3.2 If (target = a[m]) then End with output as m
   3.3 If (target < a[m]) then right = m-1
   3.4 If (target > a[m]) then left = m+1
3. End with output as none

The basic operation here is comparison.

Worst Case Analysis: **Step 3.2** to **Step 3.4** each performs **1** comparison and these steps executes **logn** times. **logn** is the number of times the while loop executes. Each time we are reducing the problem size by half. i.e **n/2, n/4, n/8, …** The loop terminates when we hit on a single element array and this would occur when the while loop has executed **logn** times. The maximum number of comparisons in Binary Search would be **3 * logn**. Hence the worst case complexity of Binary search is **O(logn)**.



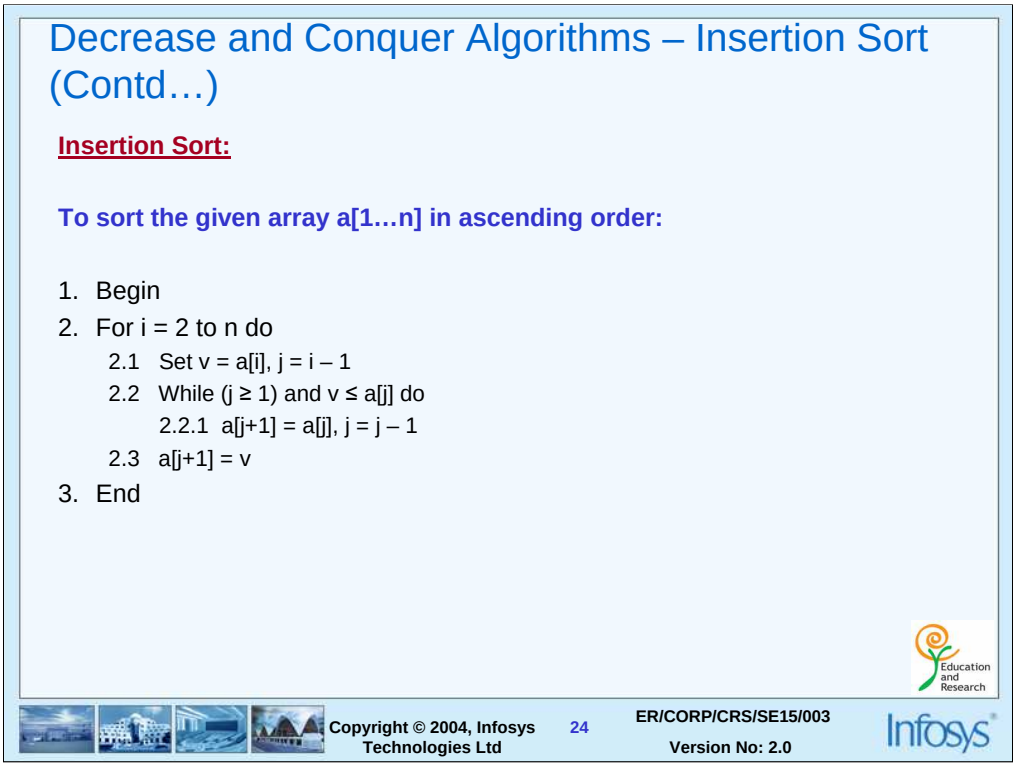
Many a time we could establish a relationship between the solution for the problem of a given size and the solution of the same problem with smaller size. Say for example $\mathbf{a^n = a^{(n-1)} * a}$. Such a relationship will help us to solve the problem easily. The Decrease and Conquer algorithm design technique is based on exploiting such relationships. The decrease in the problem size can be a constant for each iteration as in the case of the power example mentioned above or it can be a decrease by a constant factor for each iteration. For example $\mathbf{a^n = (a^{(n/2)})^2}$ (the constant factor here is 2). There can also be a variable size decrease in the problem size like in the case of **Euclid's algorithm to compute GCD, GCD(a , b) = GCD(b , a mod b)**.

We will analyze the **Insertion Sort** which works on the design principle of **Decrease and Conquer**.

Decrease and Conquer Algorithms – Insertion Sort

- Insertion sort is a sorting algorithm which works on the algorithm design principle of Decrease and Conquer
- Insertion sort works by decreasing the problem size by a constant in each iteration and exploits the relationship between the solution for the problem of smaller size and that of the original problem
- Insertion Sort is the sorting technique which we normally use in the cards game

The Insertion sort works on "*Decrease by One*" technique. In each iteration the problem size is decreased by one. Suppose **a[1…n]** is a given array to be sorted. The way insertion sort works is it assumes the smaller problem size **a[1…n-1]** is already sorted and it exploits this to put **a[n]** in place. This is done by scanning from the right most position, which is **n-1**, in **a[1…n-1]** and traversing towards the left till a value lesser than or equal to **a[n]** is encountered. Once this found we insert **a[n]** next to that value.



The basic operation here again is comparison and swapping. The analysis of the Insertion sort is as follows:

Worst Case Analysis: **Step 2.2.1** performs **1** swapping and this step is repeated at most **(i-1 – 1 + 1) = i+1** times. This is true because **j** varies from **i-1** and goes on till it is **1**. Also the number of comparisons in the while loop is **2 * (i+1)**. So the while loop performs **3 * (i+1)** basic operations. Step **2.3** does **1** swapping. So the number of operations performed by the statements inside the for loop is **3 * (i+1) + 1 (=3i+4)** and the for loop executes **n-1** times. So the total number of basic operations performed by the insertion sort is

$3(2+3+4+\ldots n) +4(n-1) = 3((n(n+1)/2) – 1) + 4(n-1) = O(n^2)$. Hence the worst case complexity of Insertion sort is **$O(n^2)$**

Average Case Analysis: The statements inside the for loop perform anywhere **1 to (3i +4)** basic operations. So on an average the number of basic operations performed by these statements is **(3i+4+1)/2 = (3i+5)/2**. Now **i** varies from **2** to **n**. So the average number of basic operations performed by Insertion sort is

$(1/2)(3(2+3+4+\ldots+n) + 5(n-1)) = (1/2)(3((n(n+1)/2)-1) + 5(n-1)) = O(n^2)$. Hence the average case complexity of Insertion sort is also **$O(n^2)$**

Dynamic Programming

- Dynamic Programming is a design principle which is used to solve problems with overlapping sub problems
- It solves the problem by combining the solutions for the sub problems
- The difference between Dynamic Programming and Divide and Conquer is that the sub problems in Divide and Conquer are considered to be disjoint and distinct various in Dynamic Programming they are overlapping.

Dynamic Programming first appeared in the context of optimization problems. The term programming here refers to the tabular method to solve the problem. Fibonacci Series is an example of a problem which has overlapping sub problems. This can be best solved using Dynamic Programming when compared to Divide and Conquer strategy.

Dynamic Programming – Computing a Binomial Coefficient

- The Binomial Coefficient, denoted by *C(n,k),* is the number of combinations of *k* elements from a set of *n* elements.
- The Binomial Coefficient is computed using the following formula

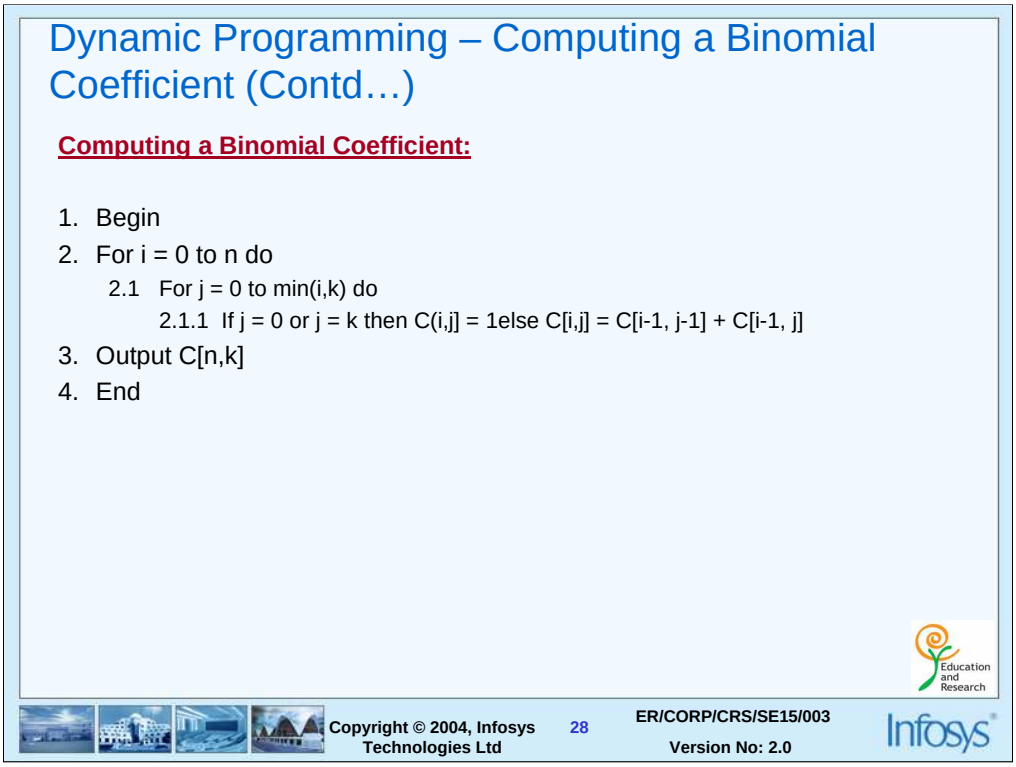
  C(n,k) = C(n-1,k-1) + C(n-1,k) for all n > k > 0 and C(n,0) = C(n,n) = 1

Computing the Binomial Coefficient is one of the standard non optimization problem for which Dynamic Programming technique is used. The recurrence relation gives us a feel of the overlapping sub problems. To compute this we tabulate values of the binomial coefficient as shown in the figure in the following slide. The values in the table are filled using the algorithm which we will introduce shortly.

Dynamic Programming – Computing a Binomial Coefficient (Contd…)

	0	1	2 . . .	k-1	k
0	1				
1	1	1			
2	1	2	1		
.					
k	1				1
.					
n-1	1			C(n-1,k-1)	C(n-1,k)
n	1				C(n,k)

C(n,k) is computed by filling the above table. We fill row by row from row 0 to n. The algorithm in the next slide fills the table with the values using Dynamic Programming technique.



The basic operation in Computing a Binomial Coefficient is addition. Let **T(n,k)** be the total number of additions done by this algorithm to compute *C(n,k)*.

The entire summation needs to be split into two parts, one covering the triangle comprising of the first *k+1* rows (refer to the table in the earlier slide) and the other covering the rectangle comprising of the remaining (*n-k)* rows.

It can be shown that the **A(n,k) = ((k-1)(k))/2 + k(n-k) = O(nk)**

Summary

- Algorithm Design Methodologies
  - Studied the various algorithm design methodologies like Brute Force, Greedy approach, Divide and Conquer, Decrease and Conquer and Dynamic Programming Techniques
- Analysis of well known algorithms
  - Analyzed certain well known algorithms which work on one of above mentioned design principles