Parul® University
Vadodara, Gujarat

NAAC GRADE A++

Information and Communication Technology

Introduction and Analysis of Algorithms

Chapter 1

Mrs. Bhumi Shah

Assistant Professor

Computer Science and Engineering

Content

1. Structure of divide-and-conquer algorithms
2. Examples:
   - Binary Search
   - Quick Sort
   - Merge Sort
3. Strassen Multiplication
4. Max-Min problem

Introduction to Divide and Conquer Algorithm

Divide and Conquer Algorithm is a problem-solving technique used to solve problems by dividing the main problem into subproblems, solving them individually and then merging them to find solution to the original problem. Divide and Conquer is mainly useful when we divide a problem into independent subproblems.

Working of Divide and Conquer Algorithm

Divide and Conquer Algorithm can be divided into three steps: Divide, Conquer and Merge.

1. Divide:

- Break down the original problem into smaller subproblems.
- Each subproblem should represent a part of the overall problem.
- The goal is to divide the problem until no further division is possible.

2. Conquer:

- Solve each of the smaller subproblems individually.
- If a subproblem is small enough (often referred to as the “base case”), we solve it directly without further recursion.
- The goal is to find solutions for these subproblems independently.

Working of Divide and Conquer Algorithm

3. Merge:

- Combine the sub-problems to get the final solution of the whole problem.
- Once the smaller subproblems are solved, we recursively combine their solutions to get the solution of larger problem.
- The goal is to formulate a solution for the original problem by merging the results from the subproblems.

Catachrestic of Divide and Conquer Algorithm

- **Dividing the Problem**: The first step is to break the problem into smaller, more manageable subproblems. This division can be done recursively until the subproblems become simple enough to solve directly.
- **Independence of Subproblems**: Each subproblem should be independent of the others, meaning that solving one subproblem does not depend on the solution of another. This allows for parallel processing or concurrent execution of subproblems, which can lead to efficiency gains.

Catachrestic of Divide and Conquer Algorithm

- **Conquering Each Subproblem**: Once divided, the subproblems are solved individually. This may involve applying the same divide and conquer approach recursively until the subproblems become simple enough to solve directly, or it may involve applying a different algorithm or technique.
- **Combining Solutions**: After solving the subproblems, their solutions are combined to obtain the solution to the original problem. This combination step should be relatively efficient and straightforward, as the solutions to the subproblems should be designed to fit together seamlessly.

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/

Parul® University
Vadodara, Gujarat

NAAC GRADE A++

Information and Communication Technology

Introduction and Analysis of Algorithms

Chapter 1

Mrs. Bhumi Shah

Assistant Professor

Computer Science and Engineering

Content

1. Structure of divide-and-conquer algorithms
2. Examples:
   - Binary Search
   - Quick Sort
   - Merge Sort
3. Strassen Multiplication
4. Max-Min problem

What is Binary Search?

Binary Search is an efficient searching algorithm that works on **sorted arrays**. It repeatedly divides the search interval in half:

- If the target is **equal to the middle** element, return it.
- If the target is **less**, search the **left half**.
- If the target is **greater**, search the **right half**.

Steps

Algorithm Steps:

1. Set two pointers: low = 0, high = n – 1
2. While low ≤ high:

- Compute mid = (low + high) / 2
- If arr[mid] == target, return index
- If arr[mid] > target, set high = mid – 1
- If arr[mid] < target, set low = mid + 1If not found, return -1

3. If not found, return -1

Binary Search Algorithm

Algorithm: Function biniter(T[1,…,n], x)

```
    if x > T[n] then return n+1
    i ← 1;
    j ← n;
    while i < j do
        k ← (i + j ) ÷ 2
        if  x ≤ T [k]  then  j ← k
        else i ← k + 1
    return i
```

Binary Search Analysis

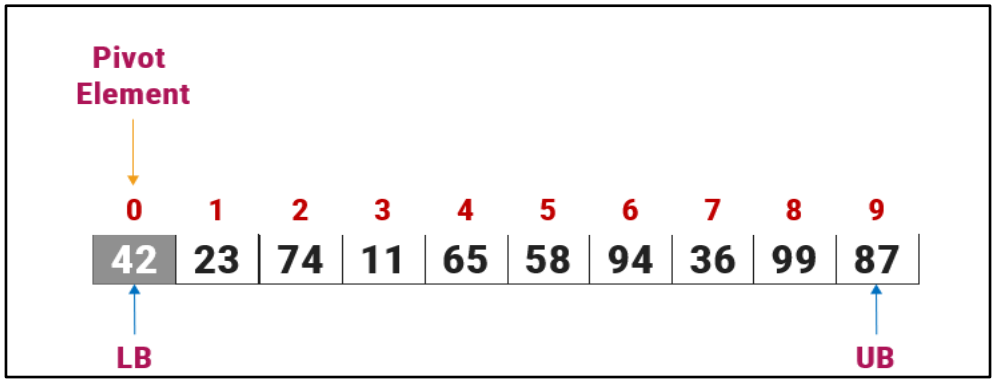
- Let $t(n)$ be the time required for a call on binrec($T[i,\ldots,j]$, x), where $n = j - i + 1$ is the number of elements still under consideration in the search.
- The recurrence equation is given as,
- $\boldsymbol{t(n)=t(n/2)+\theta(1)}$ $\boldsymbol{T(n) = aT(n/b) + f(n)}$
- Comparing this to the general template for divide and conquer algorithm, a=1, b=2 and $f(n)=\theta(1)$.
- $\boldsymbol{\therefore t(n)\in\theta(logn)}$
- The complexity of binary search is $\boldsymbol{\theta(logn)}$

Example

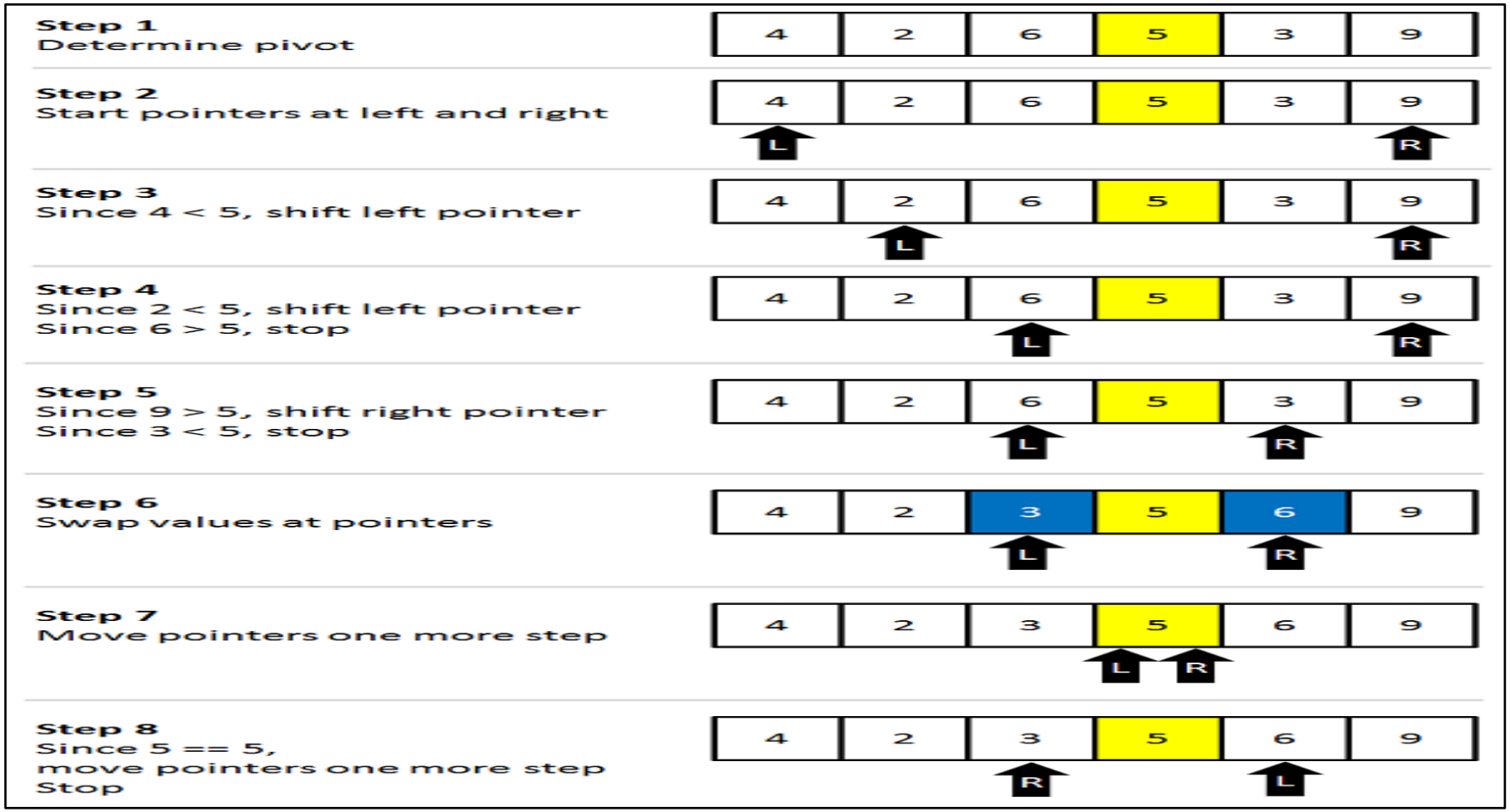
Array: [10, 20, 30, 40, 50, 60, 70]
Target: 40
Step 1:low = 0, high = 6 → mid = 3
arr[3] = 40 → FOUND
Result: Element found at index 3

Quick Sort

- Quick sort chooses the first element as a pivot element, a **lower bound is the first index** and an **upper bound is the last index**.
- The array is then **partitioned** on either side of the pivot.
- Elements are moved so that, those **greater** than the pivot are shifted to its **right** whereas the others are shifted to its **left.**
- Each Partition is **internally sorted recursively**.



Quick Sort algorithm Example



Quick Sort Algorithm

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }}
    swap(arr[i + 1], arr[high]);
    return i + 1;
}
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high); // Partitioning index
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }}
```

Quick Sort Analysis

1. Worst Case

Running time depends on **which element is chosen as key or pivot** element.

The worst case behavior for quick sort occurs when the array is partitioned into one sub-array with $\boldsymbol{n-1}$ **elements and the other with** $\boldsymbol{0}$ **element**.

In this case, the recurrence will be,

$$T(n) = T(n-1) + T(0) + \theta(n)$$

$$T(n) = T(n-1) + \theta(n)$$

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

Quick Sort Analysis

2. Best Case

   Occurs when partition produces sub-problems each of size n/2.

Recurrence equation:

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

$$l = 2, b = 2, k = 1, so\ l = b^k$$

$$\boldsymbol{T(n) = \theta(nlogn)}$$

3. Average Case

   Average case running time is much closer to the best case.

   If suppose the partitioning algorithm produces a **9:1 proportional** split the recurrence will be,

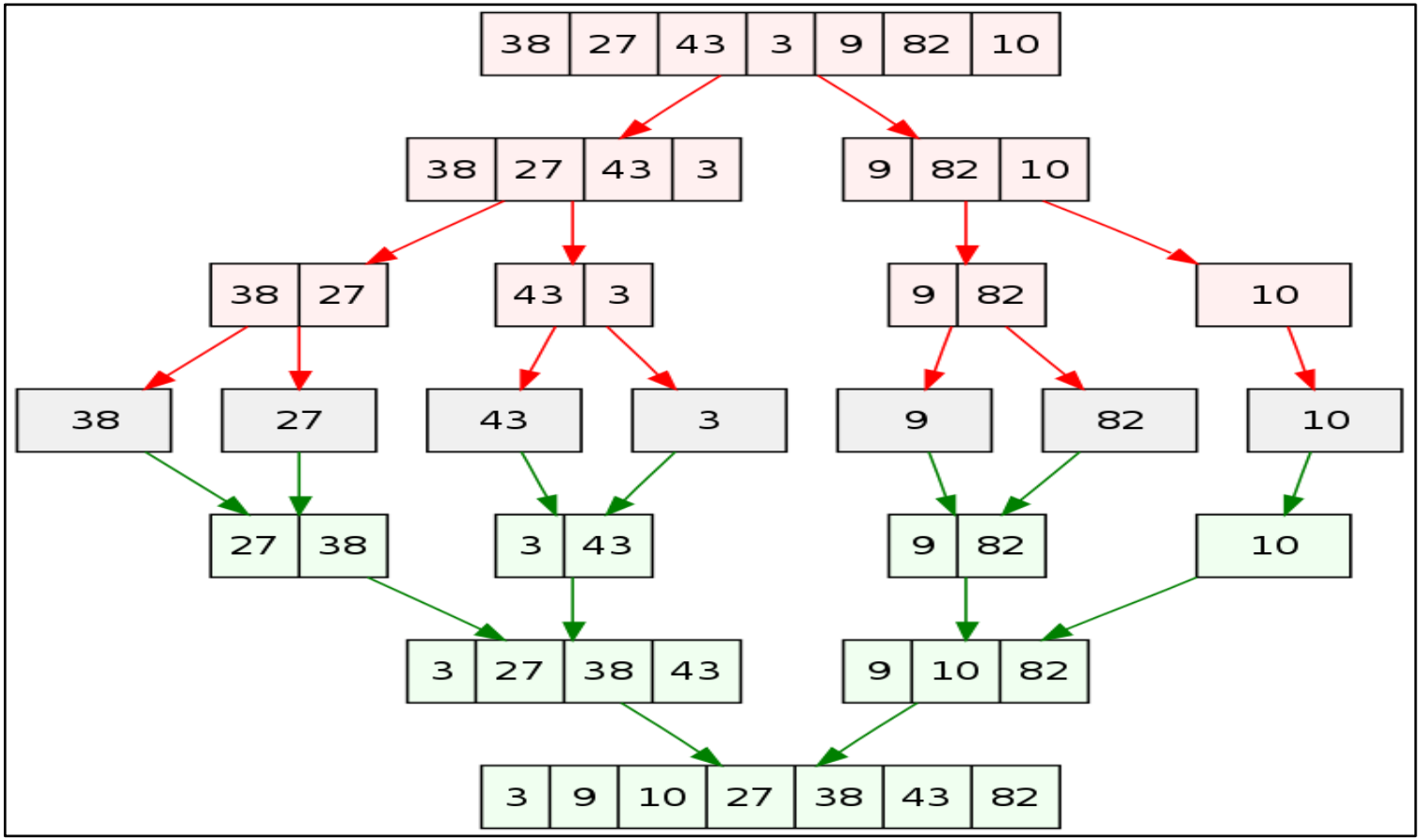
$$T(n) = T(9n/10) + T(n/10) + \theta(n)$$

$$\boldsymbol{T(n) = \theta(nlogn)}$$

Merge Sort

- Merge Sort is an example of divide and conquer algorithm.
- It is based on the idea of breaking down a list into several sub-lists until each sub list consists of a single element.
- Merging those sub lists in a manner that results into a sorted list.
- Procedure
- Divide the unsorted list into N sub lists, each containing 1 element
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2
- Repeat the process till a single sorted list of all the elements is obtained

Merge Sort Example



Merge Sort Algorithm

```
void merge(int A[ ] , int start, inmid, int end)
{
//stores the starting position of both parts in temporary variables.
int p = start ,q = mid+1;
int Arr[end-start+1] , k=0;
for(int i = start ;i <= end ;i++) {
   if(p > mid)      //checks if first part comes to an end or not .
      Arr[ k++ ] = A[ q++] ;
  else if ( q > end)   //checks if second part comes to an end or not
      Arr[ k++ ] = A[ p++ ];
  else if( A[ p ] < A[ q ])     //checks which part has smaller element.
     Arr[ k++ ] = A[ p++ ];
```

Merge Sort Algorithm

```
else
     Arr[ k++ ] = A[ q++];
 }
  for (int p=0 ; p< k ;p ++) {
   /* Now the real array has elements in sorted manner including both
       parts.*/
     A[ start++ ] = Arr[ p ] ;
  }
}
```

Merge Sort Analysis

- Let $\boldsymbol{T(n)}$ be the time taken by this algorithm to sort an array of n elements.
- Separating T into U & V takes linear time; $merge(U, V, T)$ also takes linear time.

$T(n)=T(n/2)+ T(n/2)+g(n)$ where $g(n)\in\theta(n)$.

$T(n)=2t(n/2)+ \theta(n)$ $\quad \boldsymbol{t(n) = lt(n/b) + g(n)}$

- Applying the general case, l=2, b=2, k=1
- Since $l=bk$ the second case applies so, $t(n)\in\theta(nlogn)$.
- Time complexity of merge sort is $\boldsymbol{\theta(nlogn)}$.

$$t(n) = \begin{cases} \theta(n^k) & if\ l < b^k \\ \theta(n^k logn) & if\ l = b^k \\ \theta(n^{\log_b l}) & if\ l > b^k \end{cases}$$

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/

Parul® University
Vadodara, Gujarat

NAAC GRADE A++

Information and Communication Technology

Introduction and Analysis of Algorithms

Chapter 1

Mrs. Bhumi Shah

Assistant Professor

Computer Science and Engineering

Content

1. Structure of divide-and-conquer algorithms
2. Examples:
   - Binary Search
   - Quick Sort
   - Merge Sort
3. Strassen Multiplication
4. Max-Min problem

Matrix Multiplication

Multiply following two matrices. Count how many scalar multiplications are required.

$$\begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix} . \begin{bmatrix} 6 & 8 \\ 4 & 2 \end{bmatrix}$$

$$answer = \begin{bmatrix} 1 \times 6 + 3 \times 4 & 1 \times 8 + 3 \times 2 \\ 7 \times 6 + 5 \times 4 & 7 \times 8 + 5 \times 2 \end{bmatrix}$$

To multiply 2×2 matrices, total 8 (2^3) scalar multiplications are required.

Matrix Multiplication

In general, A and B are two 2×2 matrices to be multiplied.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{21} \end{bmatrix} \text{ and } B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

Computing each entry in the product takes $\boldsymbol{n}$ **multiplications** and there are $\boldsymbol{n^2}$ **entries** for a total of $\boldsymbol{O(n^3)}$.

Strassen's Algorithm for Matrix Multiplication

- Consider the problem of **multiplying** two $n \times n$ matrices.
- Strassen's devised a better method which has the **same basic method** as the multiplication of long integers.
- The main idea is **to save one multiplication** on a small problem and then use recursion.

Strassen's Algorithm for Matrix Multiplication

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{21} \end{bmatrix} \text{ and } B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$S_1 = B_{12} - B_{22}$$
$$S_2 = A_{11} + A_{12}$$
$$S_3 = A_{21} + A_{22}$$
$$S_4 = B_{21} - B_{11}$$
$$S_5 = A_{11} + A_{22}$$
$$S_6 = B_{11} + B_{22}$$
$$S_7 = A_{12} - A_{22}$$
$$S_8 = B_{21} + B_{22}$$
$$S_9 = A_{11} - A_{21}$$
$$S_{10} = B_{11} + B_{12}$$

$$P_1 = A_{11} \cdot S_1$$
$$P_2 = S_2 \cdot B_{22}$$
$$P_3 = S_3 \cdot B_{11}$$
$$P_4 = A_{22} \cdot S_4$$
$$P_5 = S_5 \cdot S_6$$
$$P_6 = S_7 \cdot S_8$$
$$P_7 = S_9 \cdot S_{10}$$

All above operations involve only **one multiplication**.

$$P_1 = A_{11} \cdot S_1$$
$$P_2 = S_2 \cdot B_{22}$$
$$P_3 = S_3 \cdot B_{11}$$
$$P_4 = A_{22} \cdot S_4$$
$$P_5 = S_5 \cdot S_6$$
$$P_6 = S_7 \cdot S_8$$
$$P_7 = S_9 \cdot S_{10}$$

All above operations involve only **one multiplication**.

Strassen's Algorithm Analysis

It is therefore possible to multiply two 2×2 matrices using only **seven scalar multiplications**.

Let $t(n)$ be the time needed to multiply two $n \times n$ matrices by **recursive use of equations**.

$$\boldsymbol{t(n) = 7t\left(\frac{n}{2}\right) + g(n)} \qquad \boldsymbol{t(n) = lt(n/b) + g(n)}$$

Where $g(n) \in O(n^2)$.

The general equation applies with $l = 7, b = 2$ and $k = 2$.

Since $l > b^k$, the **third case** applies and $t(n) \in O(n^{lg7})$.
Since $lg7 > 2.81$, it is possible to multiply two $n \times n$ matrices in a time $\boldsymbol{O(n^{2.81})}$.

$$\boldsymbol{t(n) = \begin{cases} \theta(n^k) & if\ l < b^k \\ \theta(n^k log n) & if\ l = b^k \\ \theta(n^{log_b l}) & if\ l > b^k \end{cases}}$$

Max-Min Problem

- Finding a maximum and minimum element from a given array is the application of the Divide and Conquer algorithm.
- There are various ways to this problem, but the most traditional approach to to solve this problem is the linear approach. In the linear approach, we traverse all elements once and find the minimum and maximum element.
- In this approach, the time complexity to solve this problem is θ(n).

Example of Max-Min Problem

Max-Min Problem Algorithm

Max-Min Problem Algorithm

- Step 1: Find the mid of the array.
- Step 2: Find the maximum and minimum of the left subarray recursively.
- Step 3: Find the maximum and minimum of the right subarray recursively.
- Step 4: Compare the result of step 3 and step 4
- Step 5: Return the minimum and maximum.

Max-Min Problem Algorithm

```
MinMaxDAC(A, i, j) {
    if(i == j)
        return (A[i], A[i]);
    if((j - i) == 1)
        if (A[i] < A[j])
            return (A[i], A[j])
        else
            return (A[j], A[i])
    else {
        int mid = (i + j) / 2;
        LMin, LMax = MinMaxDAC(A, i, mid);
        RMin, RMax = MinMaxDAC(A, mid + 1, j);
```

Max-Min Problem Algorithm

```
if(LMax > RMax)
        max = LMax;
    else
        max = RMax;
    if(LMin < RMin)
        min = LMin;
    else
        min = RMin;
    return (min, max);
```

Max-Min Problem Analysis

$T(n) = 1$, if $n = 1$ or $n = 2$

$T(n) = 2T(n/2) + C$, if $n > 2$

After solving the above recurrence relation,

$T(n) \approx O(n)$

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/