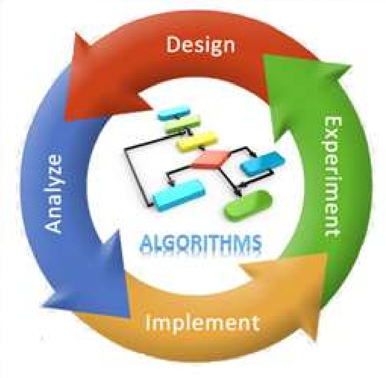
Analysis and Design of Algorithms (ADA)
GTU # 3150703

Unit-3: Divide and Conquer **Algorithms**



Rupesh G. Vaishnav
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot
rupesh.vaishnav@darshan.ac.in
9428037452

Outline

- Introduction to Recurrence Equation
- Different methods to solve recurrence
- Divide and Conquer Technique
- Multiplying large Integers Problem
- Problem Solving using divide and conquer algorithm –
  - ✓ Binary Search
  - ✓ Sorting (Merge Sort, Quick Sort)
  - ✓ Matrix Multiplication
  - ✓ Exponential

Recurrence Equation

Introduction

- Many algorithms (divide and conquer) are recursive in nature.
- When we analyze them, we get a recurrence relation for time complexity.
- We get running time as a function of n (input size) and we get the running time on inputs of smaller sizes.
- A recurrence is a recursive description of a function, or a description of a function in terms of itself.
- A recurrence relation recursively defines a sequence where the next term is a function of the previous terms.

Methods to Solve Recurrence

- Substitution
- Homogeneous (characteristic equation)
- Inhomogeneous
- Master method
- Recurrence tree
- Intelligent guess work
- Change of variable
- Range transformations

Substitution Method – Example 1

- We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

Example 1:

Time to solve the instance of size $n-1$

$$T(n) = \underline{T(n-1)} + n \quad (1)$$

Time to solve the instance of size n

- Replacing n by $n-1$ and $n-2$, we can write following equations.

$$\underline{T(n-1)} = \underline{T(n-2)} + n - 1 \quad (2)$$

$$\underline{T(n-2)} = T(n-3) + n - 2 \quad (3)$$

- Substituting equation 3 in 2 and equation 2 in 1 we have now,

$$T(n) = T(n-3) + n - 2 + n - 1 + n \quad (4)$$

Substitution Method – Example 1

$$T(n) = T(n-3) + n - 2 + n - 1 + n \quad \text{---- } 4$$

- From above, we can write the general form as,

$$T(n) = T(n-k) + (n-k+1) + (n-k+2) + \ \dots + n$$

- Suppose, if we take $k = n$ then,

$$T(n) = T(n-n) + (n-n+1) + (n-n+2) + \ \dots + n$$

$$T(n) = 0 + 1 + 2 + \ \dots + n$$

$$\boxed{T(n) = \frac{n(n+1)}{2} = O\left(n^2\right)}$$

Substitution Method – Example 2

$$t(n)=\begin{cases} c1 \; if \; n=0 \\ c2+t(n-1) \quad o/w \end{cases}$$

- Rewrite the equation, $t(n)=c2+\underline{t(n-1)}$
- Now, replace $\mathbf{n}$ by $\mathbf{n-1}$ and $\mathbf{n-2}$

$$t(n-1)=c2+\underline{t(n-2)}$$
$$\underline{t(n-2)}=c2+t(n-3)$$

$$\therefore \underline{t(n-1)}=c2+c2+t(n-3)$$

- Substitute the values of $\mathbf{n-1}$ and $\mathbf{n-2}$

$$t(n)=c2+c2+c2+t(n-3)$$

- In general,

$$t(n)=kc2+t(n-k)$$

- Suppose if we take $k=n$ then,

$$t(n)=nc2+t(n-n)=nc2+t(0)$$
$$\boxed{t(n)=nc2+c1=\boldsymbol{O(n)}}$$

Substitution Method Exercises

- Solve the following recurrences using substitution method.

1. $\mathrm{T(n)} = \begin{cases} 1 & \text{if } \mathrm{n} = 0 \text{ or } 1 \\ \mathrm{T(n-1)} + \mathrm{n} - 1 & \mathrm{o/w} \end{cases}$

2. $\mathrm{T\,(n)} = \mathrm{T\,(n-1)} + 1$ and $\mathrm{T\,(1)} = \theta\,(1)$.

$$
\begin{aligned}
T(n) &= T(n-1) + n \\
&= T(n-2) + (n-1) + n \\
&= T(n-3) + (n-2) + (n-1) + n \\
&\vdots \\
&= T(0) + 1 + 2 + \ldots + (n-2) + (n-1) + n \\
&= T(0) + \frac{n(n+1)}{2} = O(n^2)
\end{aligned}
$$

Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

Homogeneous Recurrence

- Recurrence equation

$$a_0 t_n + a_1 t_{n-1} + a_2 t_{n-2} + \cdots + a_k t_{n-k} = 0$$

- The equation of degree k in x is called the characteristic equation of the recurrence,

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k x^0$$

- Which can be factorized as,

$$p(x) = \prod_{i=1}^{k} (x - r_i)$$

- The solution of recurrence is given as,

$$\boxed{t_n = \sum_{i=1}^{k} c_i r_i^n}$$

Homogeneous Recurrence – Example 1 : Fibonacci Series

- Fibonacci series Iterative Algorithm

```
Function fibiter(n)
        i ← 1; j ← 0;
        for k ← 1 to n do
                j ← i + j;
                i ← j - i;
                return j
```

- **Analysis of Iterative Algorithm:** If we count all arithmetic operations at unit cost; the instructions inside *for* loop take constant time c. The time taken by the for loop is bounded above by n, $i.e.$, $\boldsymbol{nc} = \boldsymbol{\theta(n)}$

Case 1

Homogeneous Recurrence – Example 1 : Fibonacci Series

- If the value of n is large, then time needed to execute addition operation increases linearly with the length of operand.
- At the end of k^{th} iteration, the value of i and j will be f_{k-1} and f_k.
- As per De Moivre's formula the size of f_k **is in** $\theta(k)$.
- So, k^{th} iteration takes time in $\theta(k)$. let c be some constant such that this time is bounded above by ck for all $k \geq 1$.
- The time taken by *fibiter* algorithm is bounded above by,

$$\sum_{k=1}^{n} c.k = c.\sum_{k=1}^{n} k = c.\frac{n(n+1)}{2}$$

$$T(n) = \theta(n^2)$$

Case 2

Homogeneous Recurrence – Example 1 : Fibonacci Series

- Recursive Algorithm for Fibonacci series,

```
Function fibrec(n)
    if n < 2 then return n
    else return fibrec (n – 1) + fibrec (n – 2)
```

- The recurrence equation of above algorithm is given as,

$$T(n) = \begin{cases} n & if\ n = 0\ or\ 1 \\ T(n-1) + T(n-2) & o/w \end{cases}$$

- The recurrence can be re-written as,

$$T(n) - T(n-1) - T(n-2) = 0$$

- The characteristic polynomial is,

$$x^2 - x - 1 = 0$$

Homogeneous Recurrence – Example 1 : Fibonacci Series

- Find the roots of characteristic polynomial,

$$x^2 - x - 1 = 0$$

- The roots are,

$$r_1 = \frac{1+\sqrt{5}}{2} \text{ and } r_2 = \frac{1-\sqrt{5}}{2}$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Here, $a = 1, b = 1$ and $c = 1$

- The general solution is therefore of the form,

$$T_n = c_1 r_1^n + c_2 r_2^n$$

$$T_n = \sum_{i=1}^{k} c_i r_i^n$$

- Substituting initial values $n = 0$ and $n = 1$

$$T_0 = c_1 + c_2 = 0 \qquad (1)$$

$$T_1 = c_1 r_1 + c_2 r_2 = 1 \qquad (2)$$

- Solving these equations, we obtain

$$c_1 = \frac{1}{\sqrt{5}} \text{ and } c_2 = -\frac{1}{\sqrt{5}}$$

Homogeneous Recurrence – Example 1 : Fibonacci Series

- Substituting the values of roots and constants in general solution,

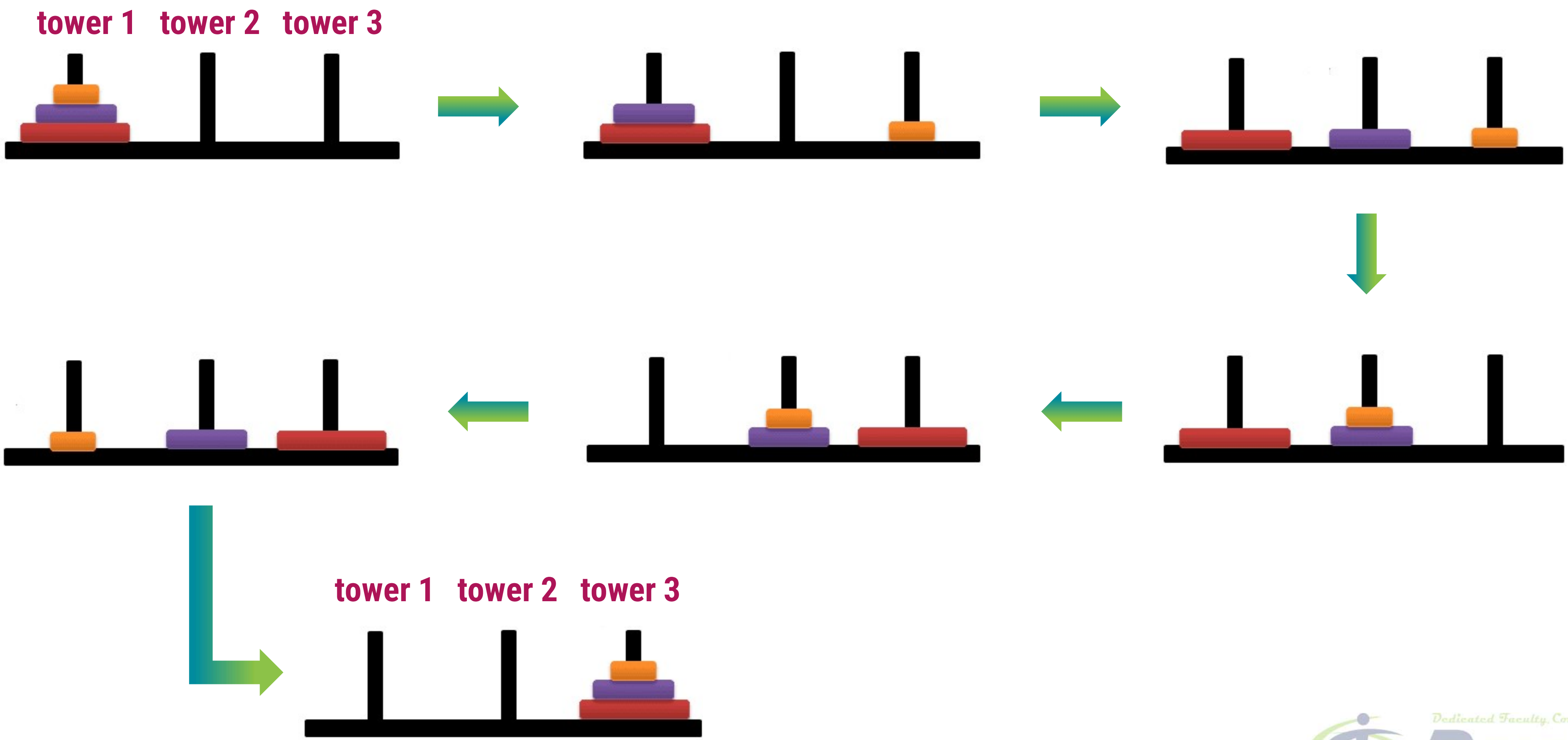
$$\boldsymbol{T_n = c_1 r_1^n + c_2 r_2^n}$$

$$T_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] \ldots\ldots\ldots \text{de Moivre's formula}$$

$$\boxed{\boldsymbol{T_n \in O(\emptyset)^n}}$$

- Time taken for recursive Fibonacci algorithm grows **Exponentially.**

Example 2 : Tower of Hanoi



Example 2 : Tower of Hanoi

- The number of movements of a ring required in the tower of Hanoi problem is given by,

$$t(m) = \begin{cases} 0 & if\ m = 0 \\ 2t(m-1)+1 & o/w \end{cases}$$

- The equation can be written as,

$$t(m) - 2t(m-1) = 1 \quad (1)$$

Inhomogeneous equation

- To convert it into a homogeneous equation, multiply with $\mathbf{-1}$ and replace $\mathbf{m}$ by $\mathbf{m-1}$,

$$-t(m-1) + 2t(m-2) = -1 \quad (2)$$

- Solving equations (1) and (2), we have now

$$t(m) - 3t(m-1) + 2t(m-2) = 0$$

Example 2 : Tower of Hanoi

$$t(m) - 3t(m-1) + 2t(m-2) = 0$$

- The characteristic polynomial is,

$$x^2 - 3x + 2 = 0$$

- Whose roots are,

$$r_1 = 2 \text{ and } r_2 = 1$$

- The general solution is therefore of the form,

$$t_m = c_1 1^m + c_2 2^m$$

- Substituting initial values $m = 0$ and $m = 1$

$$t_0 = c_1 + c_2 = 0 \quad (1)$$

$$t_1 = c_1 + 2c_2 = 1 \quad (2)$$

- Solving these linear equations we get $c_1 = -1$ and $c_2 = 1$.
- Therefore, time complexity of tower of Hanoi problem is given as,

$$\boxed{t(m) = 2^m - 1 = O(2^m)}$$

Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

Homogeneous Recurrence Exercises

- Solve the following recurrences

1. $$t_n = \begin{cases} n & if\ n = 0\ or\ 1 \\ 5t_{n-1} - 6t_{n-2} & o/w \end{cases}$$

2. $$t_n = \begin{cases} n & if\ n = 0, 1\ or\ 2 \\ 5t_{n-1} - 8t_{n-2} + 4t_{n-3} & o/w \end{cases}$$

Master Theorem

- The master theorem is a **cookbook method** for solving recurrences.
- Suppose you have a recurrence of the form

$$T(n) = aT(n/b) + f(n)$$

Time to divide & recombine

Number of sub-problems

Time required to solve a sub-problem

- **This recurrence would arise in the analysis of a recursive algorithm.**
- When input size n is large, the problem is divided up into a sub-problems each of size n/b. Sub-problems are solved recursively and results are recombined.
- The work to split the problem into sub-problems and recombine the results is $f(n)$.

Master Theorem – Example 1

$$T(n) = aT(n/b) + f(n)$$

- There are three cases:
  1. $case\ 1: if\ f(n)\ is\ in\ \boldsymbol{O}(n^{log_b a})\ [f(n) \leq n^{\log_b a}]\ then\ T(n) = \theta(n^{log_b a})$
  2. $case\ 2: f(n)\ is\ in\ \boldsymbol{\theta}(n^{log_b a})\ [f(n) = n^{\log_b a}]\ then\ T(n) = \theta(n^{log_b a}\ lgn)$
  3. $case\ 3: f(n)\ is\ in\ \boldsymbol{Ж}(n^{log_b a})\ [f(n) \geq n^{\log_b a}]\ then\ T(n) = \theta(f(n))$

- Example 1: $T(n) = 2T(n/2) + \theta(n)$ **Merge sort**
- Here $a = 2, b = 2.$ So, $n^{log_b a} = n$
- Also, $f(n) = \theta(n) = cn$
- **Case 2 applies:** $\boxed{\boldsymbol{T(n) = \theta(n\ lgn)}}$

Master Theorem – Example 2

$$T(n) = aT(n/b) + f(n)$$

- There are three cases:
  1. $case\ 1: if\ f(n)\ is\ in\ \boldsymbol{O}\left(n^{log_b a}\right)\ \left[f(n) \leq n^{\log_b a}\right]\ then\ T(n) = \theta\left(n^{log_b a}\right)$
  2. $case\ 2: f(n)\ is\ in\ \boldsymbol{\theta}\left(n^{log_b a}\right)\ \left[f(n) = n^{\log_b a}\right]\ then\ T(n) = \theta\left(n^{log_b a}\ lgn\right)$
  3. $case\ 3: f(n)\ is\ in$ **ж** $\left(n^{log_b a}\right)\ \left[f(n) \geq n^{\log_b a}\right]\ then\ T(n) = \theta(f(n))$

- Example 2: $T(n) = T(n/2) + \theta(1)$ **Binary Search**
- Here $a = 1, b = 2$. So, $n^{log_b a} = n^{\log_2 1} = n^0 = 1$
- $f(n) = \theta(1) = 1$
- **Case 2 applies: the solution is** $\boldsymbol{\theta(n^{log_b a} logn)}$
- $\boxed{\boldsymbol{T(n) = \theta(logn)}}$

Master Theorem – Example 3

$$T(n) = aT(n/b) + f(n)$$

- There are three cases:
  1. $case\ 1: if\ f(n)\ is\ in\ \boldsymbol{O}(n^{log_b a})\ [f(n) \le n^{\log_b a}]\ then\ T(n) = \theta(n^{log_b a})$
  2. $case\ 2: f(n)\ is\ in\ \boldsymbol{\theta}(n^{log_b a})\ [f(n) = n^{\log_b a}]\ then\ T(n) = \theta(n^{log_b a}\ lgn)$
  3. $case\ 3: f(n)\ is\ in$ Ж$(n^{log_b a})\ [f(n) \ge n^{\log_b a}]\ then\ T(n) = \theta(f(n))$

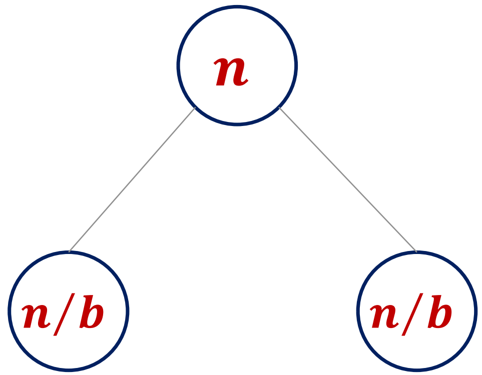
- Example 3: $T(n) = 4T(n/2) + n$
- Here $a = 4, b = 2$. So, $log_b a = 2$ and $n^{\log_b a} = n^2$
- $f(n) = n,$
- So, $f(n) \le n^2 \Rightarrow f(n)$ is in $O(n^{\log_b a})$
- **Case 1 applies:** $\boxed{\boldsymbol{T(n) = \theta(n^2)}}$

Master Theorem Exercises

- Example 4: $T(n) = 4T(n/2) + n^2$
- Example 5: $T(n) = 4T(n/2) + n^3$
- Example 6: $T(n) = 9T(n/3) + n$ (Summer 17, Summer 19)
- Example 7: $T(n) = T(2n/3) + 1$ (Summer 17)
- Example 8: $T(n) = 7T(n/2) + n^3$ (Winter 18)
- Example 9: $T(n) = 27T(n^2) + 16n$ (Winter 19)

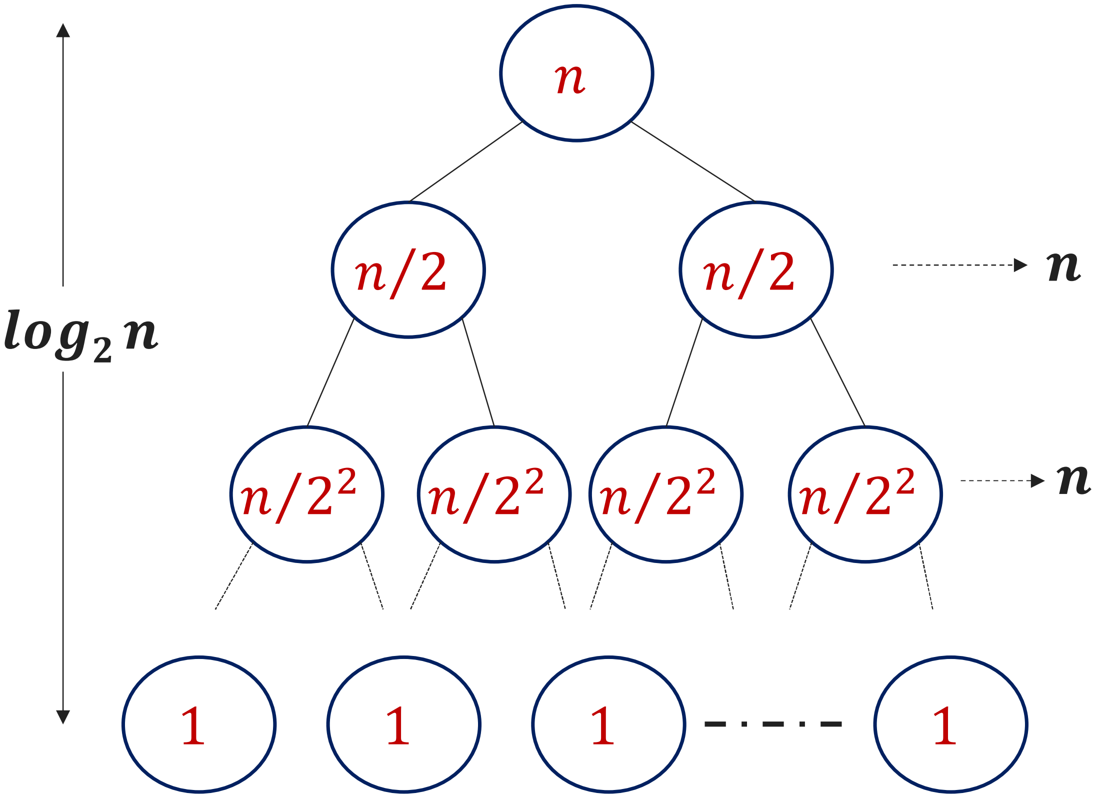
Recurrence Tree Method

- In recurrence tree, each node represents the **cost of a single sub-problem** in the set of recursive function invocations.
- We sum the **costs within each level** of the tree to obtain a set of per level costs.
- Then we sum the all the **per level costs** to determine the total cost of all levels of the recursion.
- Here while solving recurrences, we **divide the problem** into sub-problems of equal size.
- E.g., $T(n) = a\,T(n/b) + f(n)$ where $a > 1$, $b > 1$ and $f(n)$ is a given function.
- $F(n)$ is the cost of **splitting or combining** the sub problems.



Recurrence Tree Method

The recursion tree for this recurrence is



Example 1: $T(n) = 2T(n/2) + n$

- When we add the values across the levels of the recursion tree, we get a value of n for every level.
- The bottom level has $2^{\log n}$ nodes, each contributing the cost $T(1)$.

- We have $n + n + n + \dots\dots$ $\log n$ $times$

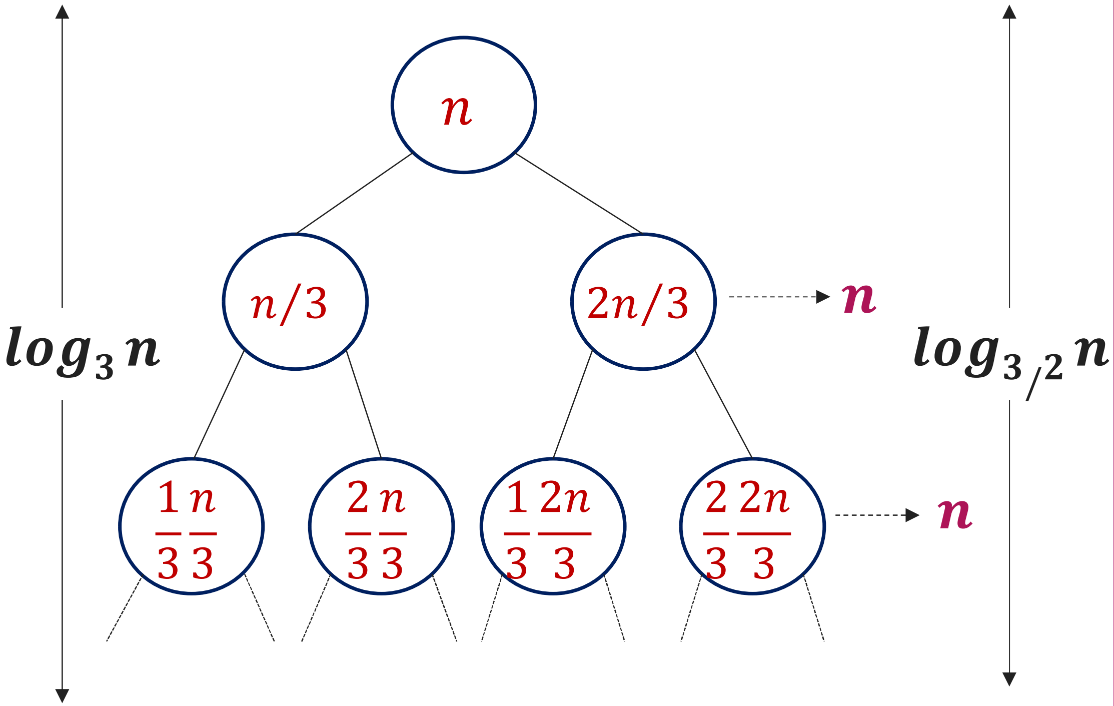
$$T(n) = \sum_{i=0}^{log_2 n-1} n + 2^{log\,n} T(1)$$

$$T(n) = n\,log\,n + n$$

$$\boxed{T(n) = O(n \log n)}$$

Recurrence Tree Method

The recursion tree for this recurrence is



Example 2: $T(n) = T(n/3) + T(2n/3) + n$

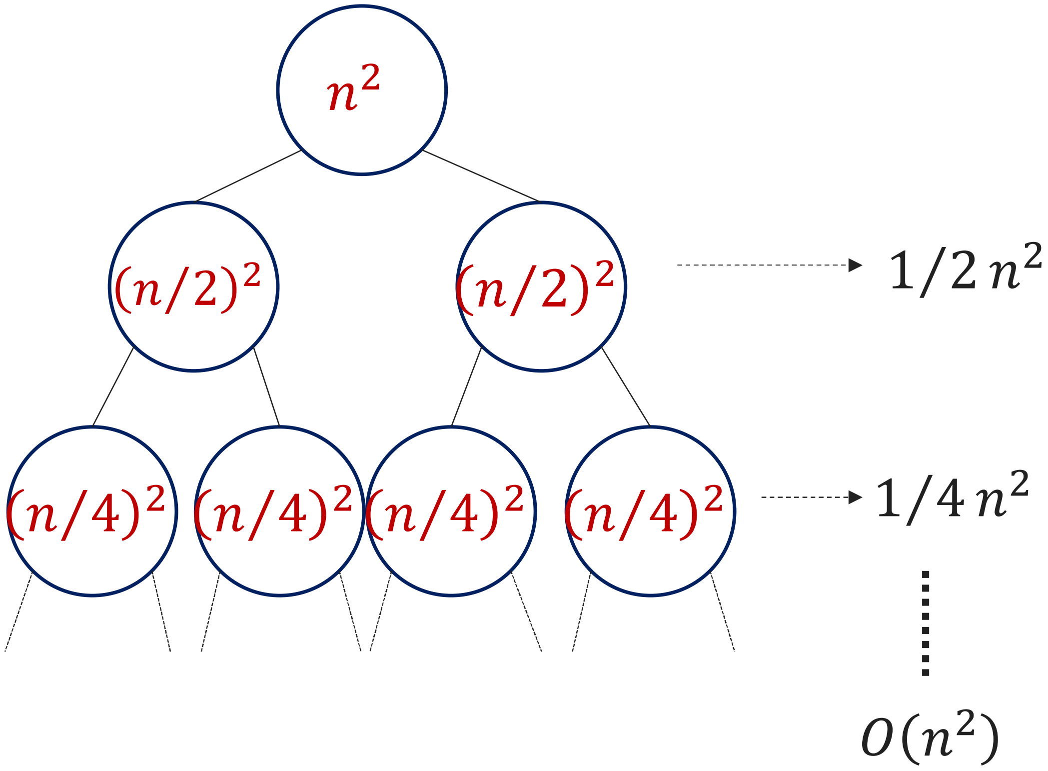
- When we add the values across the levels of the recursion tree, we get a value of n for every level.

$$T(n) = \sum_{i=0}^{\log_{3/2} n-1} n + n^{\log_{3/2} 2} T(1)$$

$$\boxed{T(n) \in n \log_{3/2} n}$$

Recurrence Tree Method

The recursion tree for this recurrence is



Example 3: $T(n) = 2T(n/2) + c.n^2$

- Sub-problem size at level i is $n/_{2^i}$
- Cost of problem at level i Is $\left(n/_{2^i}\right)^2$
- Total cost,

$$T(n) \leq n^2 \sum_{i=0}^{\log_2 n-1} \left(\frac{1}{2}\right)^i$$

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

$$T(n) \leq 2n^2$$

$$\boxed{T(n) = O(n^2)}$$

Recurrence Tree Method - Exercises

- Example 1: $T(n) = T(n/4) + T(3n/4) + c.n$
- Example 2: $T(n) = 3T(n/4) + c.n^2$
- Example 3: $T(n) = T(n/4) + T(n/2) + n^2$
- Example 4: $T(n) = T(n/3) + T(2n/3) + n$

Divide & Conquer (D&C) Technique

Introduction

- Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times.
- These algorithms typically follow a divide-and-conquer approach:
- The divide-and-conquer approach involves three steps at each level of the recursion:
  1. **Divide:** Break the problem into several sub problems that are similar to the original problem but smaller in size.
  2. **Conquer:** Solve the sub problems recursively. If the sub problem sizes are small enough, just solve the sub problems in a straightforward manner.
  3. **Combine:** Combine these solutions to create a solution to the original problem.

D&C Running Time Analysis

- The **running-time analysis** of such divide-and-conquer (D&C) algorithms is almost automatic.
- Let $g(n)$ be the **time required by D&C** on instances of size n.
- The **total time** $\boldsymbol{t(n)}$ taken by this divide-and-conquer algorithm is given by recurrence equation,

$$t(n) = lt(n/b) + g(n)$$

$$\boxed{\mathbf{T(n) = aT(n/b) + f(n)}}$$

- The solution of equation is given as,

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

where k is the power of n in $g(n)$

Binary Search

Introduction

- Binary Search is an extremely well-known instance of **divide-and-conquer** approach.
- Let $T[1 \ldots n]$ be an array of **increasing sorted order**; that is $T\,[i] \leq T[j]$ whenever $1 \leq i \leq j \leq n$.
- Let x be some number. The problem consists of **finding** $\boldsymbol{x}$ in the array T if it is there.
- If x is not in the array, then we want to find **the position** where it might be inserted.

Binary Search Example

Input: sorted array of integer values. $x = 7$

1	3	7	9	11	32	52	74	90

Step 1:

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

Find approximate midpoint

Binary Search Example

Step 2:

$x = 7$

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

Is 7 = midpoint value?

Step 3:

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

Search for the target in the area before midpoint.

Binary Search Example

Step 4:

$x = 7$

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

Find approximate midpoint

Step 5:

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

7 > value of midpoint?

Binary Search Example

Step 6:

$x = 7$

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

Search for the x in the area after midpoint.

Step 7:

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

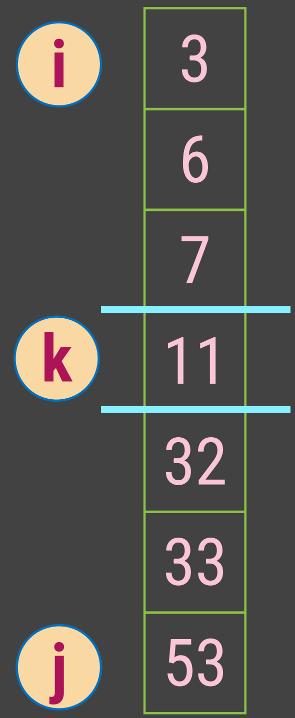
Find approximate midpoint.
Is x = midpoint value?

Binary Search – Iterative 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
```

$n = 7$

$x = 33$



Binary Search – Recursive Algorithm

```
Algorithm: Function binsearch(T[1,…,n], x)
     if n = 0 or x > T[n] then return n + 1
          else return binrec(T[1,…,n], x)
     Function binrec(T[i,…,j], x)
          if i = j then return i
          k ← (i + j) ÷ 2
          if x ≤ T[k] then
               return binrec(T[i,…,k],x)
          else return binrec(T[k + 1,…,j], x)
```

Binary Search - Analysis

- Let $t(n)$ be the time required for a call on binrec($T[i, \dots, j], x$), 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, $a = 1, b = 2 \; and \; f(n) = \theta(1)$.

$$\boldsymbol{\therefore t(n) \in \theta(\log n)}$$

- The complexity of binary search is $\boxed{\boldsymbol{\theta(\log n)}}$

- Example 2: $T(n) = T(n/2) + \theta(1)$
- Here $a = 1, b = 2$. So, $n^{log_b a} = n^{\log_2 1} = n^0 = 1$
- $f(n) = \theta(1) = 1$
- **Case 2 applies: the solution is** $\boldsymbol{\theta(n^{log_b a} log n)}$
- $\boldsymbol{T(n) = \theta(log n)}$

Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology

Binary Search – Examples

1. Demonstrate binary search algorithm and find the element $\boldsymbol{x} = \mathbf{12}$ in the following array. [3 / 4]

$$2, 5, 8, 12, 16, 23, 38, 56, 72, 91$$

2. Explain binary search algorithm and find the element $\boldsymbol{x} = \mathbf{31}$ in the following array. [7]

$$10, 15, 18, 26, 27, 31, 38, 45, 59$$

3. Let $\boldsymbol{T}[\mathbf{1}..\boldsymbol{n}]$ be a sorted array of distinct integers. Give an algorithm that can find an index $\boldsymbol{i}$ such that $\mathbf{1} \leq \boldsymbol{i} \leq \boldsymbol{n}$ and $\boldsymbol{T}[\boldsymbol{i}] = \boldsymbol{i}$, provided such an index exists. Prove that your algorithm takes time in $O(logn)$ in the worst case.

Multiplying Large Integers

Multiplying Large Integers – Introduction

- Multiplying two n digit large integers using divide and conquer method.
- Example: Multiplication of $\mathbf{981}$ by $\mathbf{1234}$.
  1. Convert both the numbers into same length nos. and split each operand into two parts:

$$\boxed{\mathbf{09|81}} \qquad \boxed{\mathbf{12|34}}$$

$$\boxed{\boldsymbol{w} = \mathbf{09}} \quad \boxed{\boldsymbol{x} = \mathbf{81}} \qquad \boxed{\boldsymbol{y} = \mathbf{12}} \quad \boxed{\boldsymbol{z} = \mathbf{34}}$$

  2. We can write as,

$$10^2w + x = 10^2(09) + 81 = 900 + 81 = 981$$

$$\boxed{\mathbf{0981} = \mathbf{10^2}\boldsymbol{w} + \boldsymbol{x}} \qquad \boxed{\mathbf{1234} = \mathbf{10^2}\boldsymbol{y} + \boldsymbol{z}}$$

Multiplying Large Integers – Example 1

- Now, the required product can be computed as,

$$
\begin{aligned}
0981 \times 1234 &= (10^2 w + x) \times (10^2 y + z) \\
&= 10^4 \underline{w \cdot y} + 10^2 \boxed{(\underline{w \cdot z} + \underline{x \cdot y})} + \underline{x \cdot z} \\
&= 1080000 + 127800 + 2754 \\
&= 1210554
\end{aligned}
$$

$w = 09$
$x = 81$
$y = 12$
$z = 34$

- The above procedure still needs **four half-size multiplications**:

$$(i) w \cdot y \quad (ii) w \cdot z \quad (iii) x \cdot y \quad (iv) x \cdot z$$

- The computation of $(\underline{w \cdot z} + \underline{x \cdot y})$ can be done as,

$$\boldsymbol{r = (w + x) \otimes (y + z) = \boxed{w \cdot y} + \boxed{(w \cdot z + x \cdot y)} + \boxed{x \cdot z}}$$

- Only **one** multiplication is required instead of two.

Additional terms

Multiplying Large Integers – Example 1

$$\boxed{10^4 w \cdot y + 10^2 (w \cdot z + x \cdot y) + x \cdot z}$$

$$\boxed{\begin{aligned} w &= 09 \\ x &= 81 \\ y &= 12 \\ z &= 34 \end{aligned}}$$

- Now we can compute the required product as follows:

$$p = w \cdot y = 09 \cdot 12 = 108$$

$$q = x \cdot z = 81 \cdot 34 = 2754$$

$$r = (w + x) \times (y + z) = 90 \cdot 46 = 4140$$

$$r = (w + x) \times (y + z) = w \cdot y + \boxed{(w \cdot z + x \cdot y)} + x \cdot z$$

$$981 \times 1234 = 10^4 p + 10^2 (r - p - q) + q$$

$$= 1080000 + 127800 + 2754$$

$$= 1210554.$$

Multiplying Large Integers – Analysis

- 981 × 1234 can be reduced to **three multiplications** of two-figure numbers (09·12, 81·34 *and* 90·46) together with a certain number of shifts, additions and subtractions.
- Reducing four multiplications to three will enable us to cut 25% of the computing time required for large multiplications.
- We obtain an algorithm that can multiply two n-figure numbers in a time,

$$T(n) = 3t\,(n/2) + g(n),$$

$$\boxed{T(n) = aT(n/b) + f(n)}$$

- Solving it gives,

$$\boxed{T(n) \in \theta\left(n^{lg3} \mid n \text{ is a power of } 2\right)}$$

Multiplying Large Integers – Example 2

- Example: Multiply **8114** with **7622** using divide & conquer method.
- Solution using D&C

Step 1: $w = 81$ $x = 14$ $y = 76$ $z = 22$

Step 2: Calculate p, q and r

$$p = w \cdot y = 81 \cdot 76 = 6156$$
$$q = x \cdot z = 14 \cdot 22 = 308$$
$$r = (w + x) \cdot (y + z) = 95 \cdot 98 = 9310$$
$$8114 \times 7622 = \mathbf{10^4 p + 10^2\ (r - p - q) + q}$$
$$= \ 61560000 + 284600 + 308$$
$$= 61844908$$

Merge Sort

Introduction

- 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

Unsorted Array

724	521	2	98	529	31	189	451
1	2	3	4	5	6	7	8

Step 1: Split the selected array

1	2	3	4	5	6	7	8
724	521	2	98	529	31	189	451

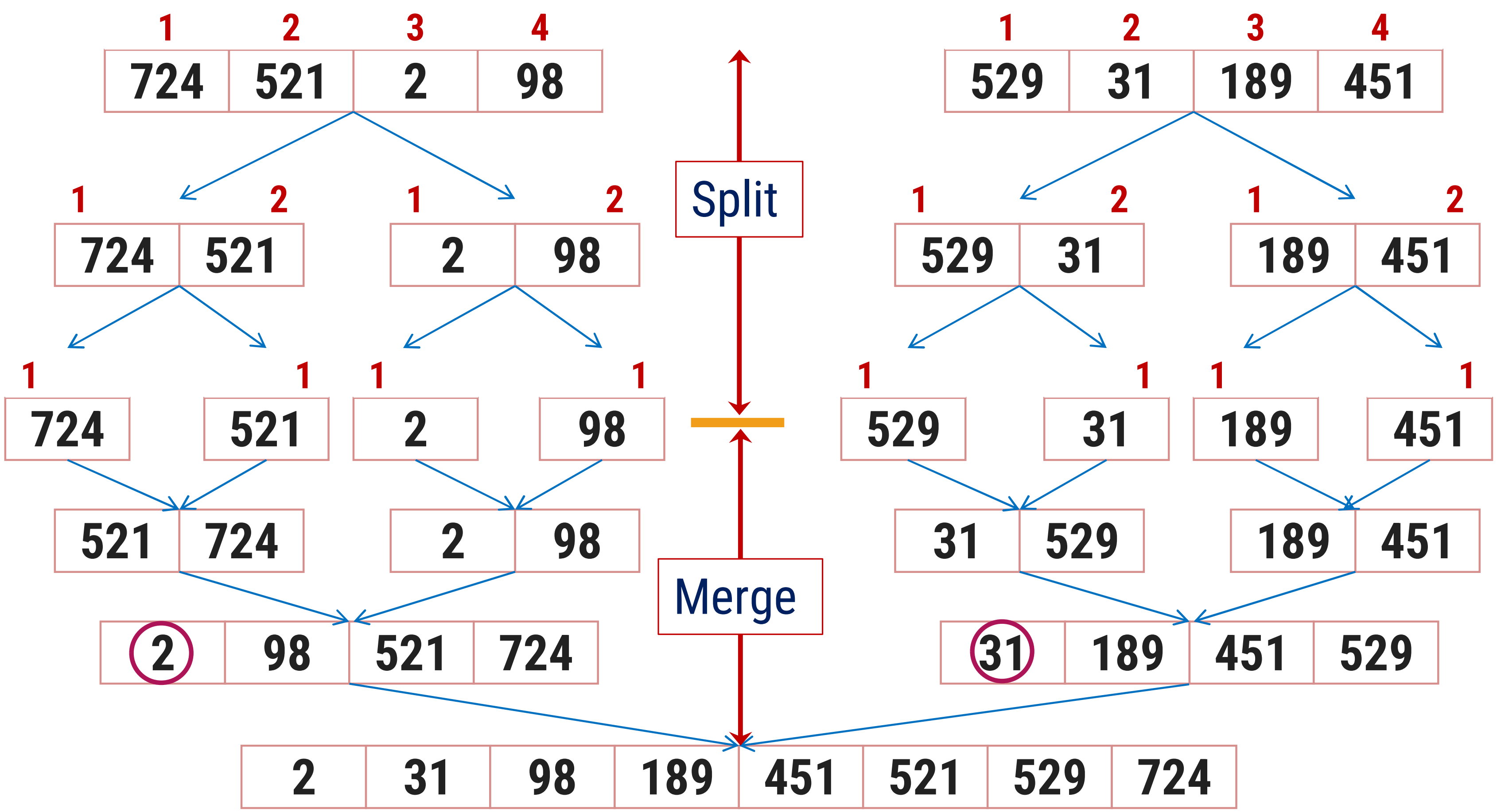
724	521	2	98
1	2	3	4

529	31	189	451
1	2	3	4

Merge Sort – Example

Select the left subarray and Split

Select the right subarray and Split



Merge Sort – Algorithm

```
Procedure: mergesort(T[1,…,n])
if n is sufficiently small then
insert(T)
else
array U[1,…,1+n/2],V[1,…,1+n/2]
      U[1,…,n/2] ← T[1,…,n/2]
      V[1,…,n/2] ← T[n/2+1,…,n]
            mergesort(U[1,…,n/2])
            mergesort(V[1,…,n/2])
            merge(U, V, T)
```

```
Procedure:
merge(U[1,…,m+1],V[1,…,n+1],T[1,…,m+n])
i ← 1;
j ← 1;
U[m+1], V[n+1] ← ∞;
for k ← 1 to m + n do
     if U[i] < V[j]
          then T[k] ← U[i];
               i ← i + 1;
     else T[k] ← V[j];
          j ← j + 1;
```

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**.

$$T(n) = T(n/2) + T(n/2) + g(n) \quad \text{where } g(n) \in \theta(n).$$

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

$$\boldsymbol{t(n) = lt(n/b) + g(n)}$$

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

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

Strassen’s Algorithm for Matrix Multiplication

Matrix Multiplication

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

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

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

- 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.

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

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

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

- 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

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

Step 1

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

Step 2

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

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

Step 3

Final Answer:

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

Where,

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_5 + P_1 - P_3 - P_7$$

No multiplication is required here.

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**.

$$\boldsymbol{t(n) = 7t(n/2) + g(n)}$$

$$\boldsymbol{t(n) = lt(n/b) + g(n)}$$

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 $t(n) \in O\left(n^{lg7}\right)$.
- Since $lg7 > 2.81$, it is possible to multiply two $n \times n$ matrices in a time $\boxed{\boldsymbol{O(n^{2.81})}}$.

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

Quick Sort

Introduction

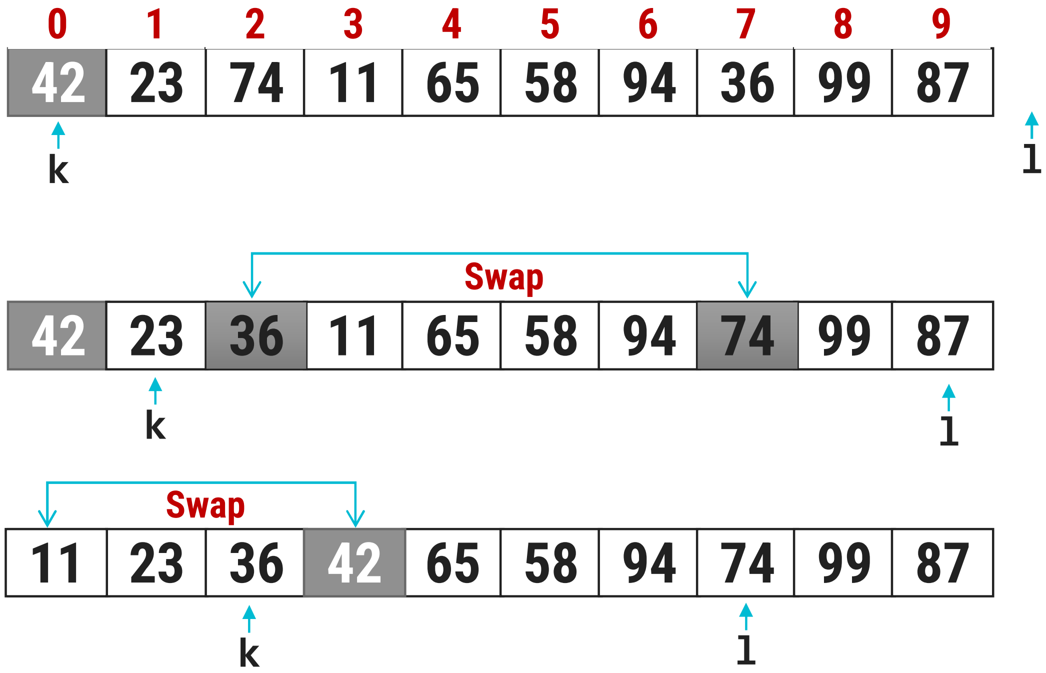
- Quick sort chooses the first element as a pivot element, a **lower bound is the first index** and an **upper bound is the last index**.
- The array is then **partitioned** on either side of the pivot.
- Elements are moved so that, those **greater** than the pivot are shifted to its **right** whereas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

Pivot Element

0	1	2	3	4	5	6	7	8	9
42	23	74	11	65	58	94	36	99	87
LB									UB

Quick Sort - Example

```
Procedure pivot(T[i,…,j]; var l)
p ← T[i]
k ← i; l ← j+1
Repeat
k ← k+1 until T[k] > p or k ≥ j
Repeat
l ← l-1 until T[l] ≤ p
While k < l do
    Swap T[k] and T[l]
    Repeat k ← k+1 until
    T[k] > p
    Repeat l ← l-1 until
    T[l] ≤ p
Swap T[i] and T[l]
```



LB = 0, UB = 9

p = 42

k = 0, l = 10

Quick Sort - Example

```
Procedure pivot(T[i,…,j]; var l)
p ← T[i]
k ← i; l ← j+1
Repeat
k ← k+1 until T[k] > p or k ≥ j
Repeat
l ← l-1 until T[l] ≤ p
While k < l do
    Swap T[k] and T[l]
    Repeat k ← k+1 until
    T[k] > p
    Repeat l ← l-1 until
    T[l] ≤ p
Swap T[i] and T[l]
```

0	1	2	3	4	5	6	7	8	9
11	23	36	42	65	58	94	74	99	87
LB		UB							

11	23	36
k		l

	LB	UB							
11	23	36	42	65	58	94	74	99	87

23	36
k	l

11	23	36	42	65	58	94	74	99	87

Quick Sort - Example

```
Procedure pivot(T[i,…,j]; var l)
p ← T[i]
k ← i; l ← j+1
Repeat
k ← k+1 until T[k] > p or k ≥ j
Repeat
l ← l-1 until T[l] ≤ p
While k < l do
    Swap T[k] and T[l]
    Repeat k ← k+1 until
    T[k] > p
    Repeat l ← l-1 until
    T[l] ≤ p
Swap T[i] and T[l]
```

				LB					UB
0	1	2	3	4	5	6	7	8	9
11	23	36	42	65	58	94	74	99	87

Swap

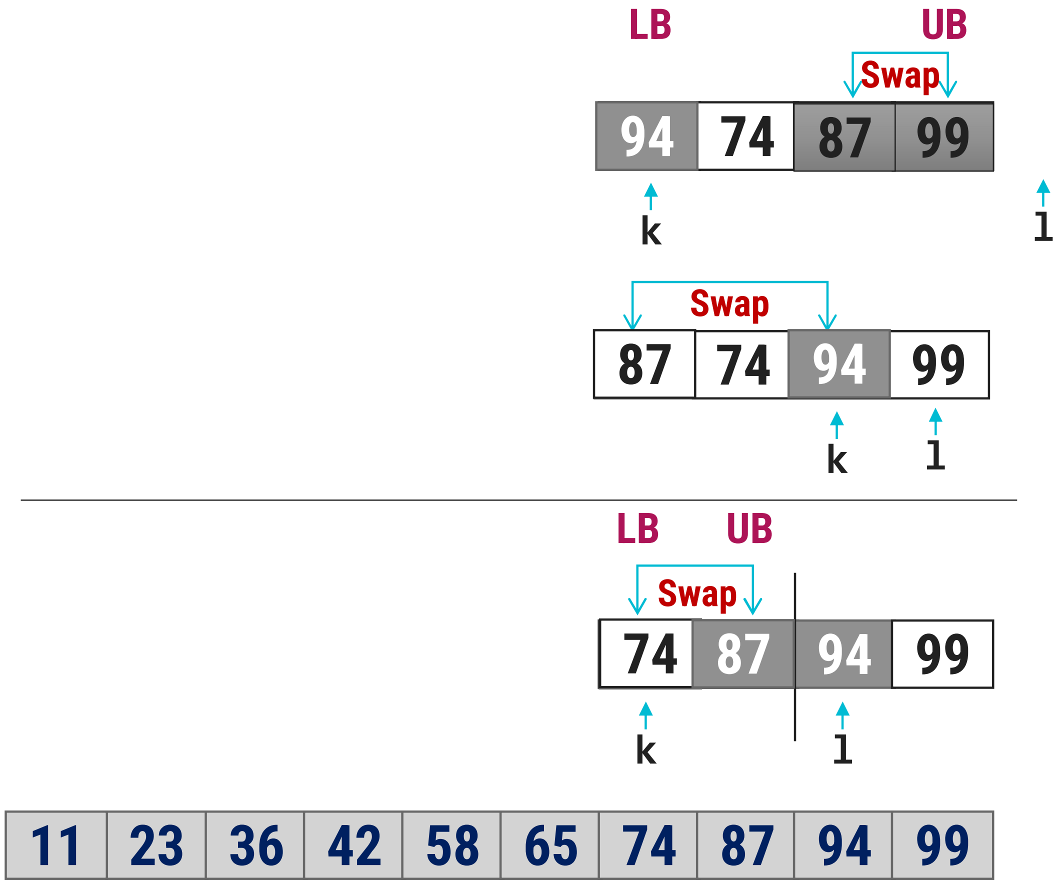
58	65	94	74	99	87
k					l

58	65	94	74	99	87

						LB			UB
11	23	36	42	58	65	94	74	99	87

Quick Sort - Example

```
Procedure pivot(T[i,…,j]; var l)
p ← T[i]
k ← i; l ← j+1
Repeat
k ← k+1 until T[k] > p or k ≥ j
Repeat
l ← l-1 until T[l] ≤ p
While k < l do
    Swap T[k] and T[l]
    Repeat k ← k+1 until
    T[k] > p
    Repeat l ← l-1 until
    T[l] ≤ p
Swap T[i] and T[l]
```



Quick Sort - Algorithm

```
Procedure: quicksort(T[i,…,j])
{Sorts subarray T[i,…,j] into
ascending order}
if j – i is sufficiently small
then insert (T[i,…,j])
else
        pivot(T[i,…,j],l)
        quicksort(T[i,…,l - 1])
        quicksort(T[l+1,…,j]
```

```
Procedure: pivot(T[i,…,j]; var l)
p ← T[i]
k ← i
l ← j + 1
repeat k ← k+1 until T[k] > p or k ≥ j
repeat l ← l-1 until T[l] ≤ p
while k < l do
        Swap T[k] and T[l]
         Repeat k ← k+1 until T[k] > p
         Repeat l ← l-1 until T[l] ≤ p
Swap T[i] and T[l]
```

Quick Sort Algorithm – 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,

$$T(n) = T(n-1) + T(0) + \theta(n)$$
$$T(n) = T(n-1) + \theta(n)$$
$$\boxed{\boldsymbol{T(n) = \theta(n^2)}}$$

2. Best Case
   - **Occurs when partition produces sub-problems each of size n/2.**
   - Recurrence equation:

$$T(n) = 2T(n/2) + \theta(n)$$
$$l = 2, b = 2, k = 1, so\ l = b^k$$
$$\boxed{\boldsymbol{T(n) = \theta(nlogn)}}$$

Quick Sort Algorithm – Analysis

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,

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

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

Quick Sort - Examples

- Sort the following array in ascending order using quick sort algorithm.
  1. 5, 3, 8, 9, 1, 7, 0, 2, 6, 4
  2. 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9
  3. 9, 7, 5, 11, 12, 2, 14, 3, 10, 6

Exponentiation

Exponentiation - Sequential

- Let a and n be two integers. We wish to compute the **exponentiation** $\boldsymbol{x = a^n}$.
- Algorithm using **Sequential Approach:**

```
function exposeq(a, n)
   r ← a
   for i ← 1 to n - 1 do
      r ← a * r
   return r
```

- This algorithm takes a time in $\boldsymbol{\theta(n)}$ since the instruction $\boldsymbol{r = a * r}$ is executed exactly $\boldsymbol{n-1}$ **times**, provided the multiplications are counted as elementary operations.

Exponentiation - Sequential

- But **to handle larger operands**, we must consider the time required for each multiplication.
- Let m is the size of operand a.
- Therefore, the multiplication performed the i^{th} **time** round the loop concerns **an integer of size** m and **an integer whose size is between** $im - i + 1$ **and** im, which takes a time between

$$M(m, im - i + 1) \text{ and } M(m, im)$$

$a = 5$ $so\ m = 1$ *and* $n = 25$ and suppose $i = 10$

The body of loop executes 10^{th} time as,

$$r = a * r$$

here 9 times multiplication is already done so $r = 5^9 = 1953125$

The size of r in the 10th iteration will be between $im - i + 1\ to\ im$, i.e., between $1\ to\ 10$

10-10+1

10

Exponentiation - Sequential

- The total time $T(m, n)$ spent multiplying when computing an with ***exposeq*** is therefore,

$$\sum_{i=1}^{n-1} M(m, im - 1 + 1) \leq T(m, n) \leq \sum_{i=1}^{n-1} M(m, im)$$

$$T(m, n) \leq \sum_{i=1}^{n-1} M(m, im) \leq \sum_{i=1}^{n-1} cm\ im$$

$$cm^2 \sum_{i=1}^{n-1} i \leq cm^2 n^2 = \boldsymbol{\theta(m^2 n^2)}$$

- If we use the **divide-and-conquer** multiplication algorithm,

$$\boxed{\boldsymbol{T(m, n) \in \theta(m^{lg3} n^2)}}$$

Exponentiation – D & C

- Suppose, we want to compute $\boldsymbol{a^{10}}$
- We can write as,

$$a^{10} = (a^5)^2 = (a.a^4)^2 = (a.(a^2)^2)^2$$

- In general,

$$a^n = \begin{cases} a & if\ n = 1 \\ \left(a^{n/2}\right)^2 & if\ n\ is\ even \\ a \times a^{n-1} & otherwise \end{cases}$$

- Algorithm using **Divide & Conquer Approach:**

```
function expoDC(a, n)
   if n = 1 then return a
   if n is even then return [expoDC(a, n/2)]²
   return a * expoDC(a, n - 1)
```

Exponentiation – D & C

Number of operations performed by the algorithm is given by,

$$N(n) = \begin{cases} 0 & if\ n = 1 \\ N(n/2) + 1 & if\ n\ is\ even \\ N(n-1) + 1 & otherwise \end{cases}$$

Time taken by the algorithm is given by,

$$T(m,n) = \begin{cases} 0 & if\ n = 1 \\ T(m, n/2) + M(m\,n/2, m\,n/2) & if\ n\ is\ even \\ T(m, n-1) + M(m, (n-1)m) & otherwise \end{cases}$$

Solving it gives, $\boxed{\boldsymbol{T(m,n)\ \in\ \theta\ (m^{lg3}n^{lg3})}}$

```
function expoDC(a, n)
   if n = 1 then return a
   if n is even then return [expoDC(a, n/2)]²
   return a * expoDC(a, n - 1)
```

Exponentiation – Summary

	Multiplication	
	Classic	D&C
exposeq	$\theta(m^2n^2)$	$\theta(m^{lg3}n^2)$
expoDC	$\theta(m^2n^2)$	$\theta(m^{lg3}n^{lg3})$

Thank You

Dedicated Faculty, Committed Education
Darshan
Institute of Engineering & Technology