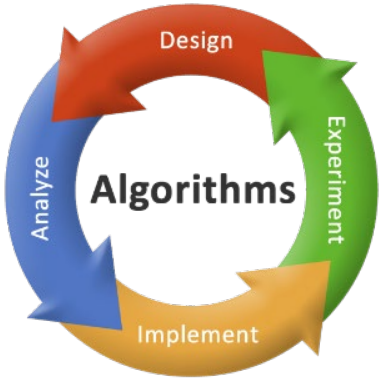
2CS503 Design & Analysis of Algorithm



Recurrence Relation

Topics to be covered

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

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.

Methods to Solve Recurrence

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

Substitution Method

- We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.
- Example 1:

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

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

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

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

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

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

Substitution Method

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

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

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

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

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

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

$$T(n) = \frac{n(n+1)}{2} = \boldsymbol{O(n^2)}$$

Exercise 1

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

Rewrite, $t(n) = c2 + t(n-1)$

Now, replace n by $n - 1$ and $n - 2$

$$t(n-1) = c2 + t(n-2)$$
$$t(n-2) = c2 + t(n-3)$$

So,

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

In general,

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

Suppose if we take $k = n$ then,

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

Exercise 2

- Solve the following recurrence using substitution method.

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

Solving recurrence relation

- Iteration (already discussed in previous slids)
- Characteristics root
- Generating function

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,

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k x^0$$ {degree=n-(n-k)=>k i.e. difference between highest and lowest}

- Which can be factorized as,

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

- The solution of recurrence is given as,

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

Analysis: Fibonacci series

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

Iterative Algorithm for Fibonacci series: 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, $i.e.$, $\boldsymbol{nc} = \boldsymbol{\theta(n)}$

Analysis: 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,

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

Analysis: Fibonacci series

- The characteristic polynomial is,

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

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

- Whose roots are,

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

- The general solution is therefore of the form,

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

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

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

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

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

Analysis: Fibonacci series

- Solving these equations, we obtain

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

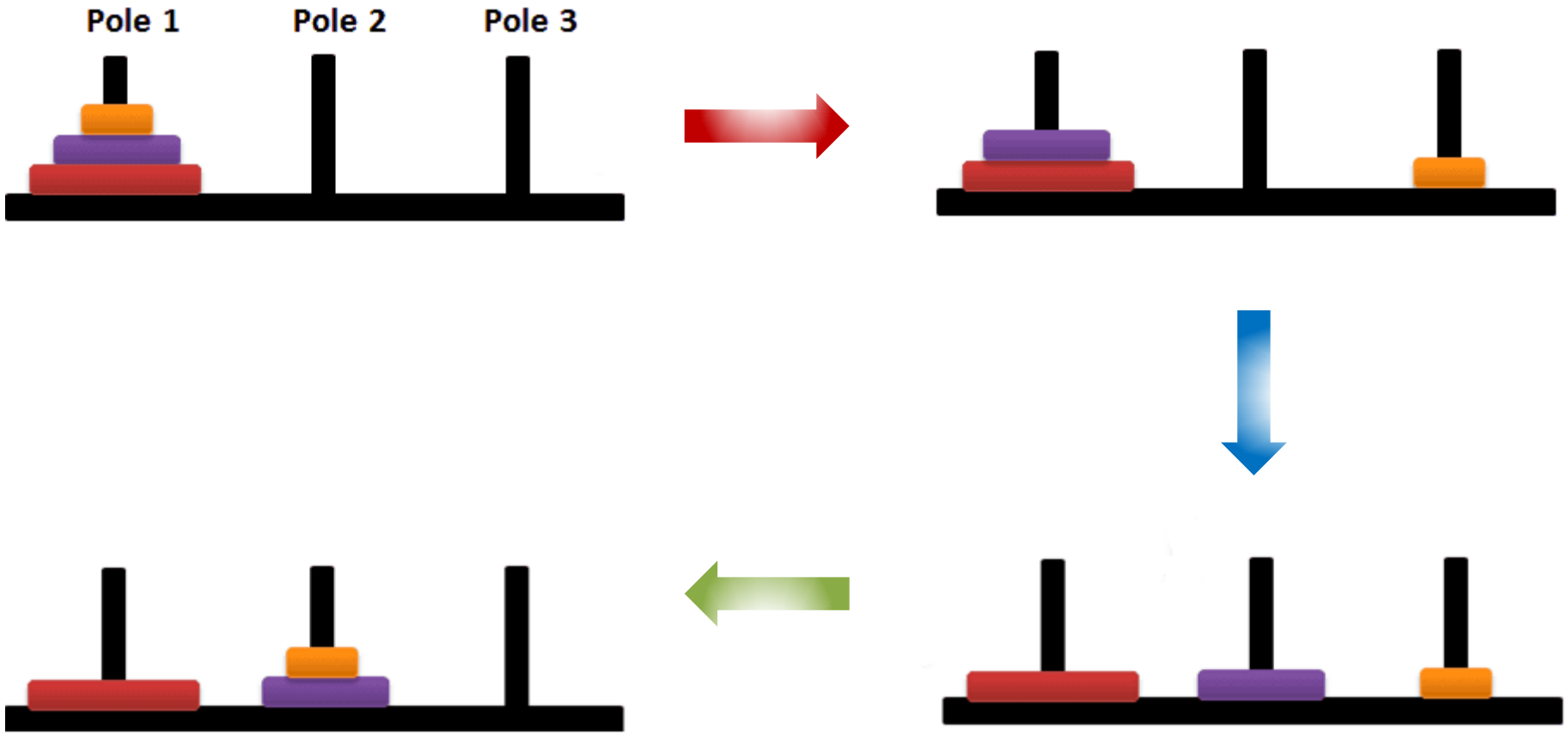
- Substituting the values of roots and constants,

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

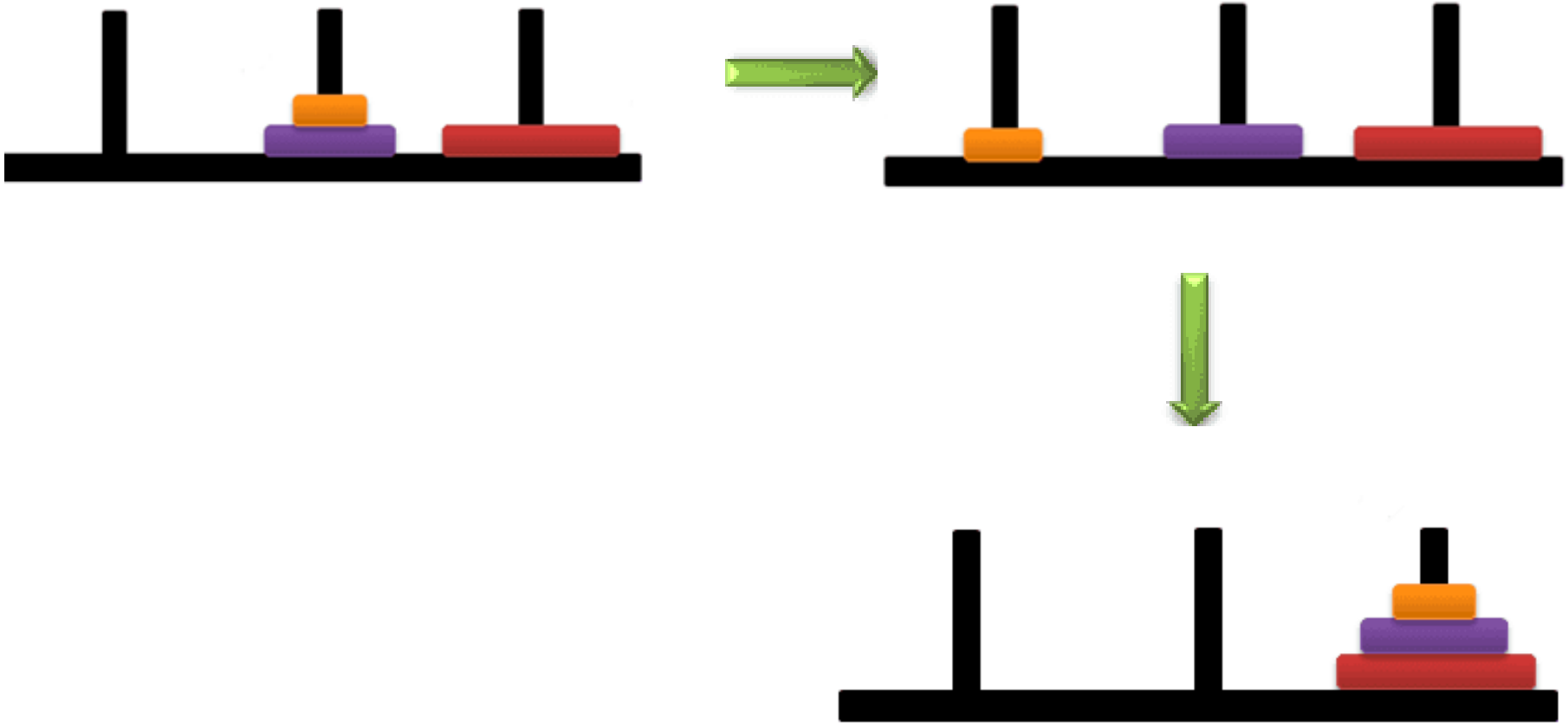
$$\boldsymbol{T_n \in O(\emptyset)^n}$$

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

Analysis: Tower of Hanoi



Analysis: Tower of Hanoi



Analysis: 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,

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

- To convert it into homogeneous equation,
  - Multiply with -1 and replace m by m-1,

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

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

$$\boldsymbol{t(m) - 3t(m-1) + 2t(m-2) = 0}$$

Analysis: Tower of Hanoi

- 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 $n = 0$ and $n = 1$

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

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

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

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

Non-homogeneous recurrence

Recurrence equation

$$a_0T(n) + a_1T(n-1) + a_2T(n-2) + \cdots + a_kT(n-k) = f(n)$$

is called as recurrence of order k, where $a_0, a_1, a_2, \ldots a_k$ are constants, it is also known as k^{th} order linear recurrence relation.

The solution of a given recurrence relation is given by:

Total solution or general solution := Homogeneous solution + Particular solution

$$\mathbf{T(n) = T(n)^h + T(n)^p}$$

Homogeneous solution($T(n)^h$): we calculate homogeneous solution by putting f(n)=0, and simply solves it.

$\mathbf{T(n)^h}$ => $a_0T(n) + a_1T(n-1) + a_2T(n-2) + \cdots + a_kT(n-k) = 0$

Particular solution($T(n)^p$): it exist only when RHS of recurrence relation is non-zero. i.e. $f(n) \neq 0$.

If $f(n) \neq 0$ we consider three possibility:

- Case1: f(n) is constant
- Case2: f(n) is polynomial of degree n
- Case3: f(n) is exponential

Particular solution($T(n)^p$)

Case1: f(n) is constant

1. find p by putting T(n)=p,
2. If fails, put T(n)=np
3. If fails, put T(n)=n^2p…

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

=> To calculate Particular solution

=> put T(n)=P
=> P-2(P-1)+P-2=1
=> 0=1 fail

=> put T(n)=nP
=> nP-2(n-1)P+(n-2)P=1
=> 0=1 fail

=> put T(n)=n^2P
=> n^2P – 2$(n-1)^2$P+(n-2)P=1
=> p=1/2
=> Now put the value of P in solution
$T(n)^p$=n^2P=>n^2/2

Particular solution($T(n)^p$)

Case2: f(n) is polynomial

1. $T(n)=d_0+d_1n+d_2n^2...d_mn^m$
2. Find d0, d1, d2.. by comparing the co-efficient

$T(n)-8T(n-1)=14n+5$

=> To calculate Particular solution

=> put $T(n)= d_0+d_1n$
=> $d_0+d_1n - 8(d_0+d_1(n-1))=14n+5$
=> $-7d_0 - 7d_1n+8d_1=5 + 14n$
=> By comparing the coefficient of n^0, n^1 term
=> $-7d_1=14$ => $d_1=-2$
=> $-7d_0+8d_1=5$ =>$d_0= -3$
Now put the value of d_0 and d_1 to get the solution
$T(n)^p=d_0+d_1n$
$T(n)^p= -3 + (-2)n$

Particular solution($T(n)^p$)

Case3: f(n) is exponential function

find d by putting $T(n)=d*a^n$,

if homogeneous solution contains a term a^k then $T(n)= n(d*a^n)$

if contain two times then

$T(n)= n^2(d*a^n)$

$T(n)-8T(n-1) =5 * 2^n$

=> To calculate Particular solution

=> put $T(n)= da^n$ {a=2}

=> $d*2^n - 8*d2^{n-1}=5*2^n$

=> $d*2^n - 4*2^1*d2^{n-1}=5*2^n$

=> $d*2^n - 4*d2^n=5*2^n$

=>$2^n(d-4d)=5*2^n$

=> By comparing the coefficient of 2^n

=> d=-5/3

Now put the value of d to get the solution

$T(n)^p=da^n$

$T(n)^p= -5/3* 2^n$

Find the total solution: T(n)-8T(n-1) =14n+5

1. Homogeneous solution($T(n)^h$)

   $T(n)-8T(n-1)=0$

   characteristic equation

   $X-8=0$, roots=(8)

   $T(n)^h = c_1 8^n$

2. Particular solution($T(n)^p$)

- **=>** put $T(n)= d_0+d_1 n$
- => $d_0+d_1 n - 8(d_0+d_1(n-1))=14n+5$
- => $-7d_0 - 7d_1 n+8d_1=5 + 14n$
- => By comparing the coefficient of n^0, n^1 term
- => $-7d_1=14$ => $d_1=-2$
- => $-7d_0+8d_1=5$ =>$d_0= -3$
- Now put the value of d_0 and d_1 to get the solution
- $T(n)^p=d_0+d_1 n$
- **$T(n)^p= -3 + (-2)n$**

Total solution($T(n)$)=($T(n)^h$) + ($T(n)^h$)

$T(n) = c_1 8^n -3 + (-2)^n$

Master Method(for divide and conquer recurrence)

- All divide and conquer algorithm divide the problem into sub-problems, each of which is part of the original problem, and then perform some additional work to compute the final answer..
- 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 Method(for divide and conquer recurrence)

As an example, a merge sort algorithm operates on two sub-problems, each of which is half the size of the original, and then performs O(n) additional work for merging. This gives the running time equation:

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

The following theorem can be used to determine the running time of divide and conquer algorithms. For a given program (algorithm), first we try to find the recurrence relation for the problem. If the recurrence is of the below form, then we can directly give the answer without fully solving it. If the recurrence is of the form

$$T(n) = aT(n/b) + \theta(nklogpn)$$

Master Method(for divide and conquer recurrence)

$$T(n) = aT(n/b) + \theta(nklogpn)$$

where a ≥ 1, b >1, k ≥ 0 and p is a real number, then:

1) If a > b^k, then $T(n) = \Theta\left(n^{log_b^a}\right)$
2) If a= b^k
   a. If $p > -1$, then $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$
   b. If $p = -1$, then $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$
   c. If $p < -1$, then $T(n) = \Theta\left(n^{log_b^a}\right)$
3) If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

1) If a > b^k, then $T(n) = \Theta\left(n^{log_b^a}\right)$

2) If a= b^k

   a. If $p > -1$, then $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$

   b. If $p = -1$, then $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$

   c. If $p < -1$, then $T(n) = \Theta\left(n^{log_b^a}\right)$

3) If $a < b^k$

   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$

   b. If $p < 0$, then $T(n) = O(n^k)$

1. $T(n) = 3T\,(n/2) + n^2$

$T(n) = n^{\log_2 3} \times n^{2-\log_2 3}$

$h(n) = \frac{n^2}{n^{\log_2 3}}$

$U(1)\ (n^2)$

1) If a > b^k, then $T(n) = \Theta\left(n^{log_b^a}\right)$

2) If a= b^k

   a. If $p > -1$, then $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$

   b. If $p = -1$, then $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$

   c. If $p < -1$, then $T(n) = \Theta\left(n^{log_b^a}\right)$

3) If $a < b^k$

   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$

   b. If $p < 0$, then $T(n) = O(n^k)$

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

$a = 1$

$b = 2$

$h(n) = \frac{n^2}{n^2} = 1$

$U(n) = O(n^2) \log_2^2 = 0$

$n^{\log_2^2} \times n^2 = \Theta(n^2)$

1) If a > b^k, then $T(n) = \Theta\left(n^{log_b^a}\right)$

2) If a= b^k

   a. If $p > -1$, then $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$

   b. If $p = -1$, then $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$

   c. If $p < -1$, then $T(n) = \Theta\left(n^{log_b^a}\right)$

3) If $a < b^k$

   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$

   b. If $p < 0$, then $T(n) = O(n^k)$

3. $T(n) = 16T(n/4) + n$

$a = 16 \quad b = 4$

$h(n) = \frac{n}{n^{\log_4 16}} = \frac{1}{n}$

$\therefore T(n) = n^{\log_4 16} \times 1$

$T(n) = n^2$

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

1) If a > b^k, then $T(n) = \Theta(n^{log_b^a})$

2) If a= b^k

   a. If $p > -1$, then $T(n) = \Theta(n^{log_b^a} log^{p+1} n)$

   b. If $p = -1$, then $T(n) = \Theta(n^{log_b^a} loglogn)$

   c. If $p < -1$, then $T(n) = \Theta(n^{log_b^a})$

3) If $a < b^k$

   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$

   b. If $p < 0$, then $T(n) = O(n^k)$

4. $T(n) = 2T(n/2) + nlogn$

$a = 1 \quad b = 2 \quad h(n) = \frac{n \log(n)}{n^1}$

$h(n) = \log n$

$\therefore U(n) = \frac{\log_2 n^2}{2}$

$O\left(\frac{n(\log n)^2}{2}\right)$

1) If a > b^k, then $T(n) = \Theta(n^{log_b^a})$

2) If a= b^k

   a. If $p > -1$, then $T(n) = \Theta(n^{log_b^a} log^{p+1} n)$

   b. If $p = -1$, then $T(n) = \Theta(n^{log_b^a} loglogn)$

   c. If $p < -1$, then $T(n) = \Theta(n^{log_b^a})$

3) If $a < b^k$

   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$

   b. If $p < 0$, then $T(n) = O(n^k)$

5. $T(n) = 2T(n/2) + n/logn$

$a = b = 2$

$h(n) = \frac{n}{\log n \times n^1}$

$\frac{1}{\log n} = (\log n)^{-1}$

$= \log n^{-1}$

$T(n) = (\log$

1) If a > b^k, then $T(n) = \Theta(n^{log_b^a})$

2) If a= b^k

   a. If $p > -1$, then $T(n) = \Theta(n^{log_b^a} log^{p+1} n)$

   b. If $p = -1$, then $T(n) = \Theta(n^{log_b^a} loglogn)$

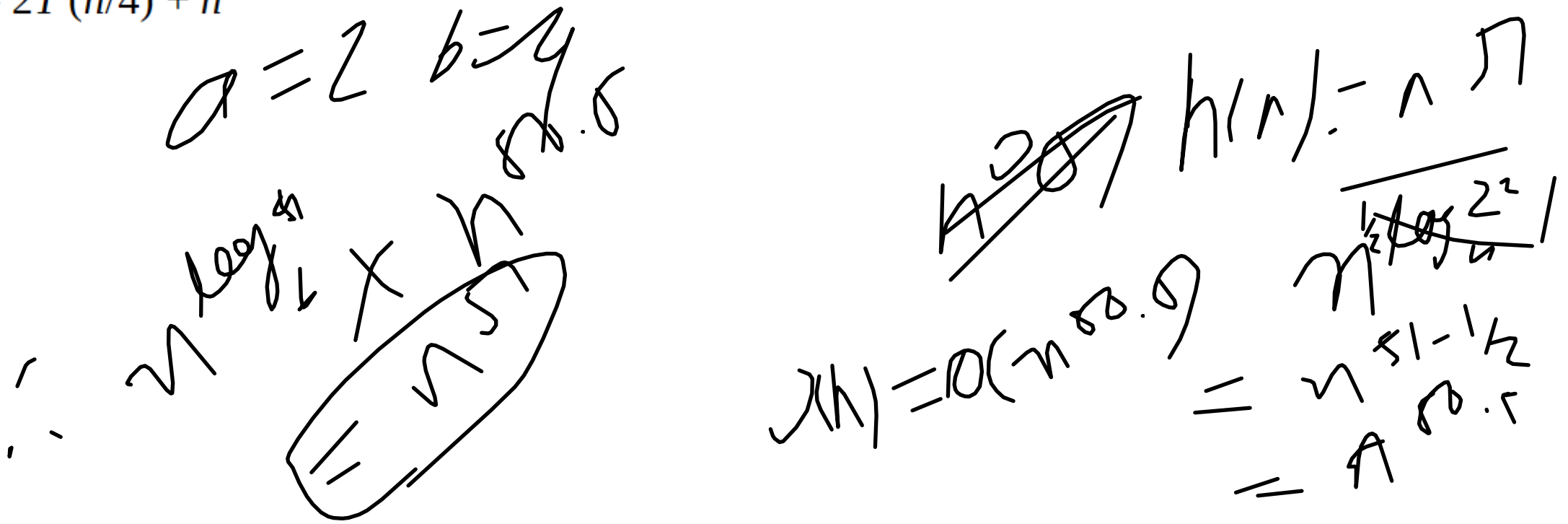
   c. If $p < -1$, then $T(n) = \Theta(n^{log_b^a})$

3) If $a < b^k$

   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$

   b. If $p < 0$, then $T(n) = O(n^k)$

6. $T(n) = 2T(n/4) + n^{051}$



1) If a > b^k, then $T(n) = \Theta\left(n^{log_b^a}\right)$

2) If a= b^k

   a. If $p > -1$, then $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$

   b. If $p = -1$, then $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$

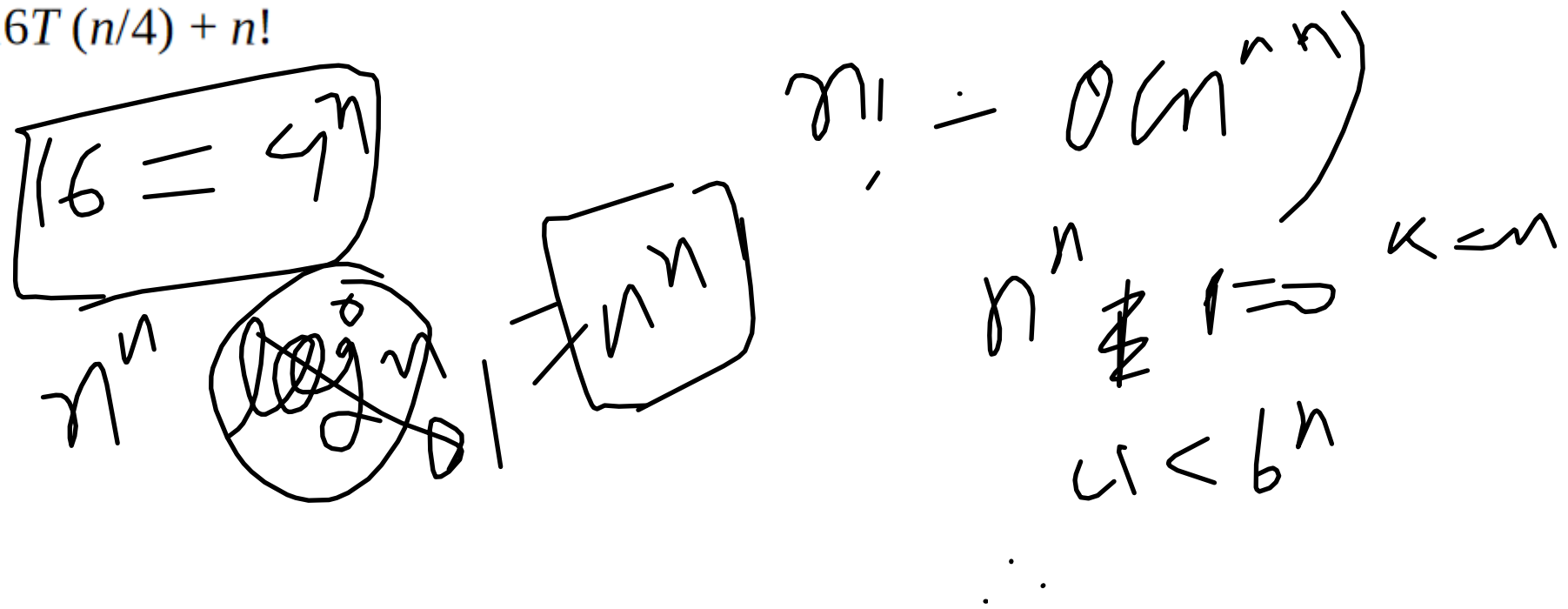
   c. If $p < -1$, then $T(n) = \Theta\left(n^{log_b^a}\right)$

3) If $a < b^k$

   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$

   b. If $p < 0$, then $T(n) = O(n^k)$

7. $T(n) = 16T\,(n/4) + n!$



1) If a > b^k, then $T(n) = \Theta(n^{log_b^a})$

2) If a= b^k

   a. If $p > -1$, then $T(n) = \Theta(n^{log_b^a} log^{p+1} n)$

   b. If $p = -1$, then $T(n) = \Theta(n^{log_b^a} loglogn)$

   c. If $p < -1$, then $T(n) = \Theta(n^{log_b^a})$

3) If $a < b^k$

   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$

   b. If $p < 0$, then $T(n) = O(n^k)$

8. $T(n) = \sqrt{2}\,T(n/2) + logn$

$a = \sqrt{2}$ $b = 2$

$\therefore \sqrt{2} > 2^0 = 1$

$T(n) = (n^{\frac{1}{2}} \log$

$h(n) = \frac{\log n}{n^{\frac{1}{2}} \log 2^{\frac{1}{2}}}$

$= \log n \times n^{-\frac{1}{2}}$

1) If a > b^k, then $T(n) = \Theta(n^{log_b^a})$

2) If a= b^k

   a. If $p > -1$, then $T(n) = \Theta(n^{log_b^a} log^{p+1} n)$

   b. If $p = -1$, then $T(n) = \Theta(n^{log_b^a} loglogn)$

   c. If $p < -1$, then $T(n) = \Theta(n^{log_b^a})$

3) If $a < b^k$

   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$

   b. If $p < 0$, then $T(n) = O(n^k)$

9. $T(n) = 3T(n/3) + \sqrt{n}$

$a = b = 3$ $h(n) = \frac{n^{1/2}}{n} = n^{-1/2}$

$\therefore O(n)$

$\sqrt{h} = O(1)$

1) If a > b^k, then $T(n) = \Theta\left(n^{log_b^a}\right)$

2) If a= b^k

   a. If $p > -1$, then $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$

   b. If $p = -1$, then $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$

   c. If $p < -1$, then $T(n) = \Theta\left(n^{log_b^a}\right)$

3) If $a < b^k$

   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$

   b. If $p < 0$, then $T(n) = O(n^k)$

10. $T(n) = 2^n T(n/2) + n^n$ $T(n) = 0.5T(n/2) + 1/n$ $T(n) = 64T(n/8) - n^2 logn$

We can not apply master method in such cases

Master Method

- Example 4: $T(n) = 4T(n/2) + n^2$
- Example 5: $T(n) = 4T(n/2) + n^3$
- Example 6: $T(n) = 9T(n/3) + n$
- Example 7: $T(n) = T(2n/3) + 1$
- Example 8: $T(n) = 7T(n/2) + n^3$
- Example 9: $T(n) = 27T(n^2) + 16n$

Master Theorem for Subtract and Conquer Recurrences

Let T(n) be a function defined on positive n, and having the property

$$T(n) = \begin{cases} c & if\ n \leq 1 \\ aT(n-b) + f(n) & if\ n > 1 \end{cases}$$

for some constants c,a $>$ 0,b $\geq$ 0,k $\geq$ 0, and function f(n). If f(n) is in $O(n^k)$, then

$$T(n) = \begin{cases} O(n^k) & if\ a < 1 \\ O(n^{k+1}) & a = 1 \\ O(n^k a^{n/b}) & if\ a > 1 \end{cases}$$

T(n) = T(n-1) + n(n-1)
a=1, b=1, k=2
Since a=1, T(n) = $O(n^{k+1})$
T(n)= $O(n^{2+1})$
T(n)= $O(n^3)$

Recurrence Tree Method

Steps to solve recurrence relations using recursion tree method:

Step-01: Draw a recursion tree based on the given recurrence relation.

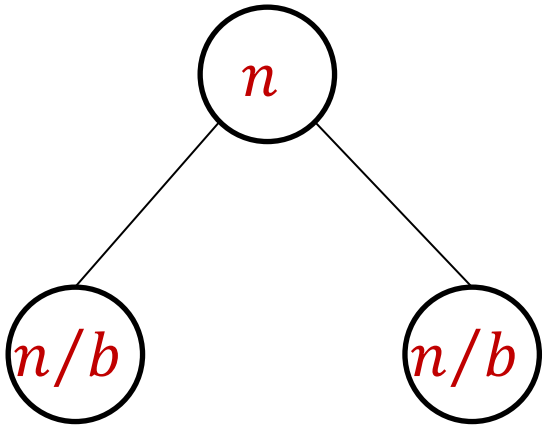
Step-02: Determine-

- Cost of each level
- Total number of levels in the recursion tree
- Number of nodes in the last level
- Cost of the last level

Step-03: Add cost of all the levels of the recursion tree and simplify the expression to obtained in terms of asymptotic notation

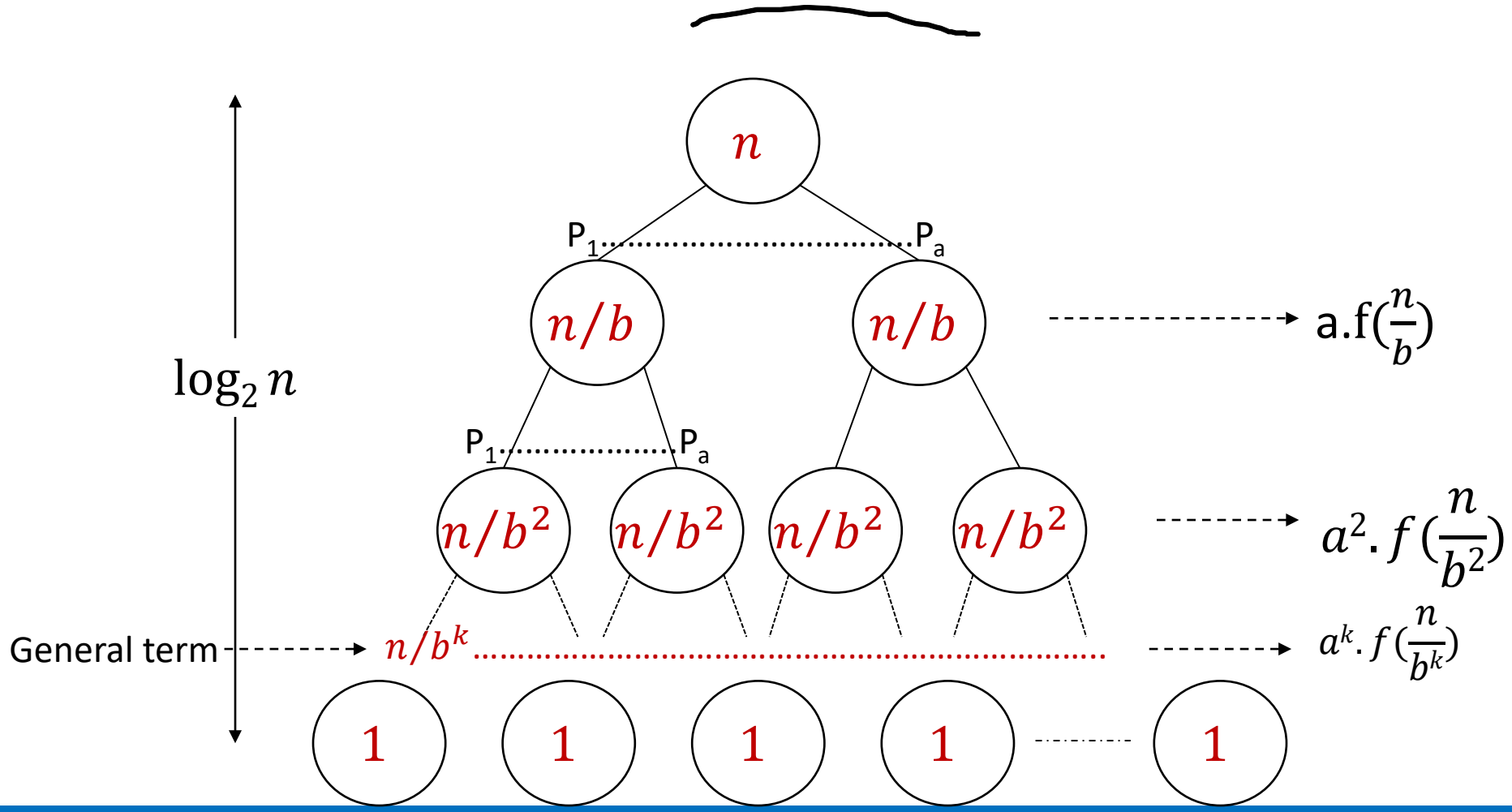
Recurrence Tree Method

- 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

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



Total time complexity

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

$$T(n) = a^k T(n/b^k) + \sum_{i=0}^{k-1} a^i f(n)$$

$n/b^k = 1$,for small input of size 1

$$n = bk$$

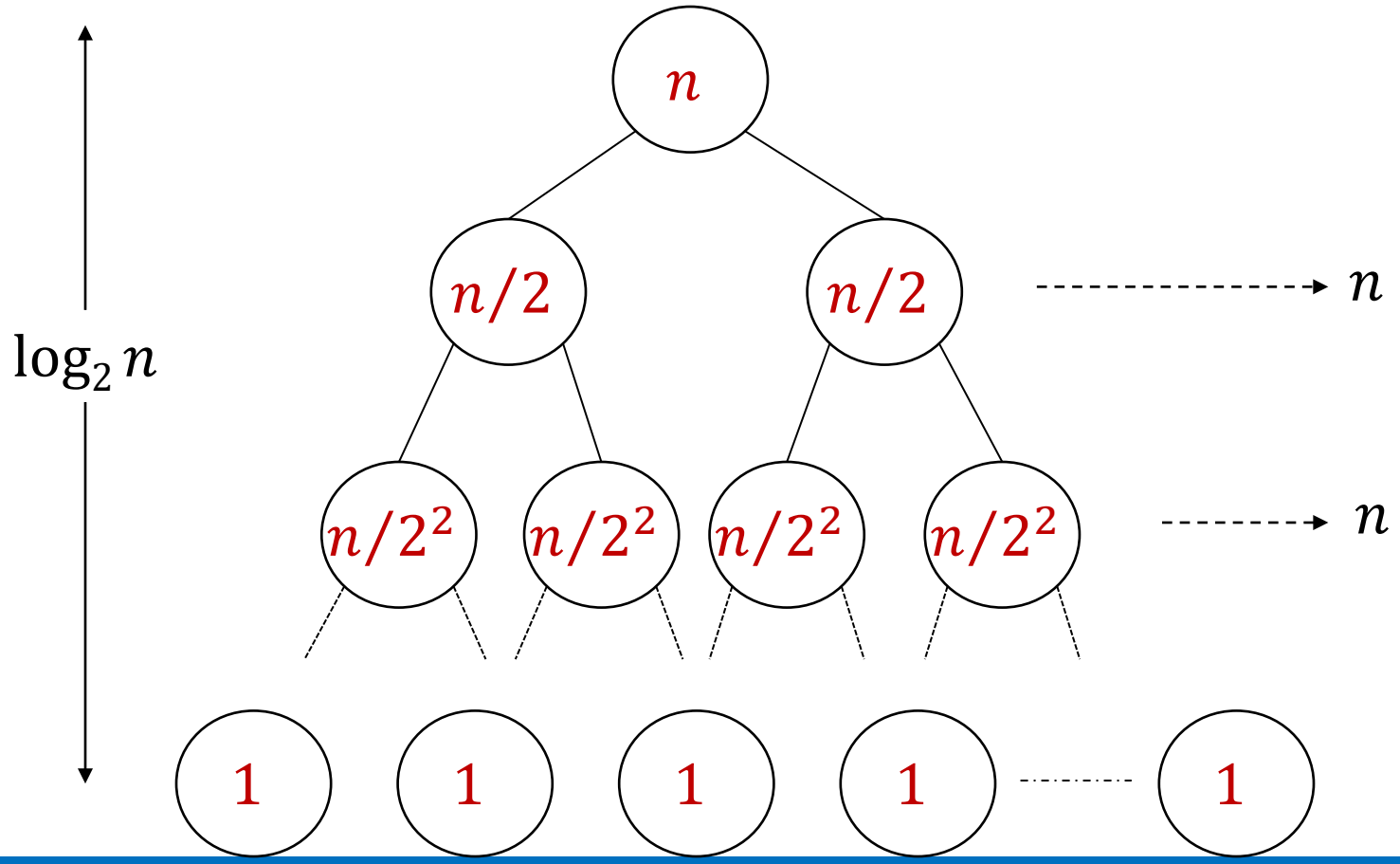
$k = \log n$ *(depth of recursion)*

$$T(n) = a^{\log_b n} T(1) + \sum_{i=0}^{\log n - 1} a^i f(n)$$

$$T(n) = n^{\log_b a} \cdot T(1) + \sum_{i=0}^{\log n - 1} a^i f(n)$$

Recurrence Tree Method

- Example 1: $T(n) = 2T(n/2) + n$
- The recursion tree for this recurrence is :



Step-01:

- Draw a recursion tree based on the given recurrence relation.
- The given recurrence relation shows-
- A problem of size n will get divided into 2 sub-problems of size n/2.
- Then, each sub-problem of size n/2 will get divided into 2 sub-problems of size n/4 and so on.
- At the bottom most layer, the size of sub-problems will reduce to 1.

The given recurrence relation shows-

- The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n.
- The cost of dividing a problem of size n/2 into its 2 sub-problems and then combining its solution is **n/2** and so on.

Step-02:

- Determine cost of each level-
- Cost of level-0 = n
- Cost of level-1 = n/2 + n/2 = n
- Cost of level-2 = n/4 + n/4 + n/4 + n/4 = n and so on.

Step-03:

- Determine total number of levels in the recursion tree-

- Size of sub-problem at level-0 = $n/2^0$
- Size of sub-problem at level-1 = $n/2^1$
- Size of sub-problem at level-2 = $n/2^2$
- Continuing in similar manner, we have- Size of sub-problem at level-i = $n/2^i$
- Suppose at level-x (last level), size of sub-problem becomes 1. Then- $n / 2^x = 1$
- $2^x = n$
- Taking log on both sides, we get- $x\log_2 2 = \log_2 n$
- $x = \log_2 n$
- ∴ Total number of levels in the recursion tree = $\log_2 n + 1$

Step-04:

- Determine number of nodes in the last level-
  - Level-0 has 2^0 nodes i.e. 1 node
  - Level-1 has 2^1 nodes i.e. 2 nodes
  - Level-2 has 2^2 nodes i.e. 4 nodes
- Continuing in similar manner, we have- Level-$\log_2 n$, has $2^{\log_2 n}$ nodes i.e. n nodes

Step-05: Determine cost of last level-

Cost of last level = n x T(1) = θ(n)

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = \underbrace{\{\ n + n + n + \ldots\ldots\ \}}_{\text{For } \log_2 n \text{ levels}} + \theta\ (n)$$

$= n \times \log_2 n + \theta\ (n)$

$= n\log_2 n + \theta\ (n)$

$= \boldsymbol{\theta\ (n\log_2 n)}$

Recurrence Tree Method

- 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 $n + n + n + \ldots\ldots \quad \log n \; times$

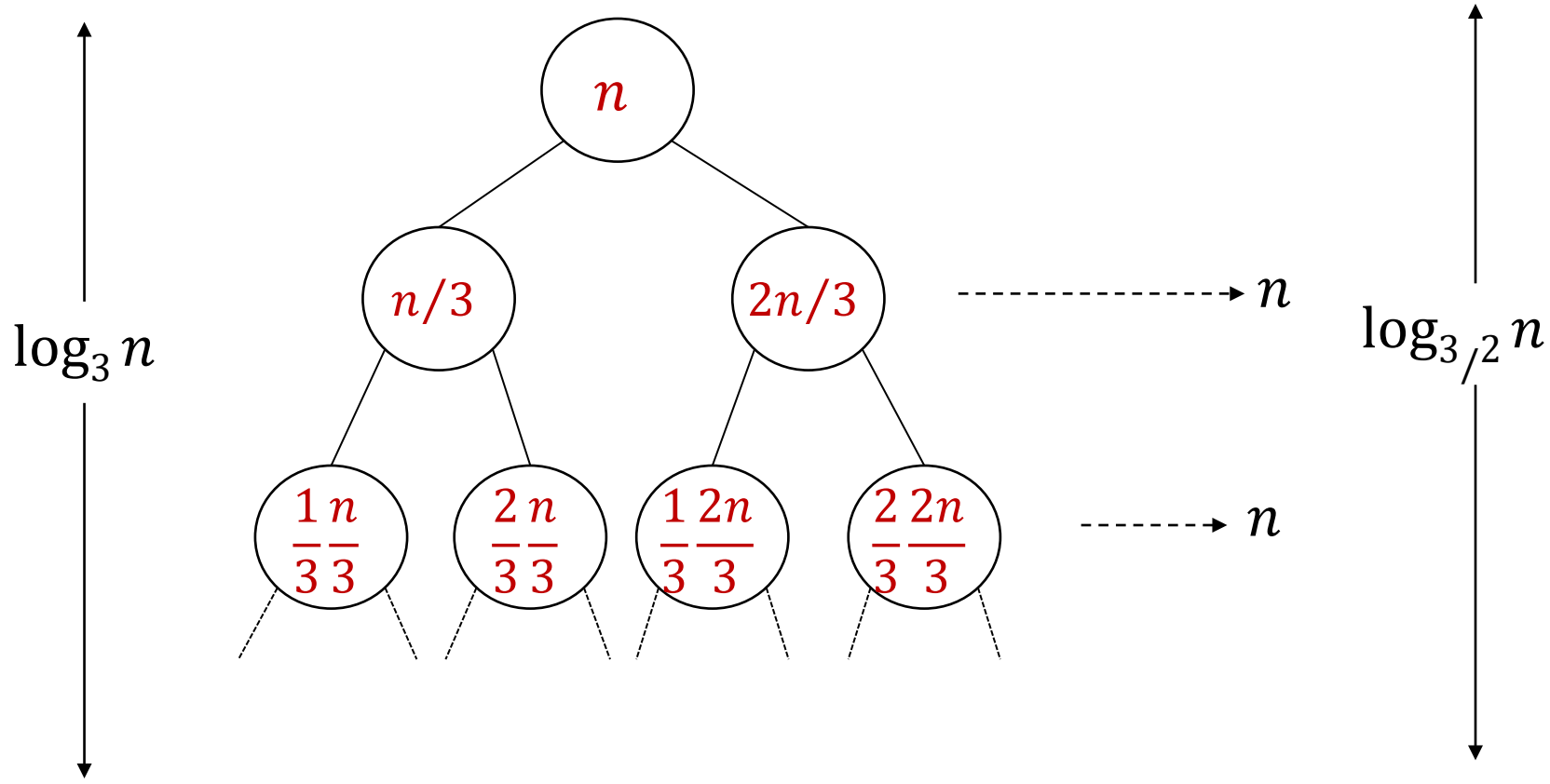
$$T(n) = \sum_{i=0}^{\log_2 n - 1} n + 2^{\log n} T(1)$$

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

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

Recurrence Tree Method

- Example 2: $T(n) = T(n/3) + T(2n/3) + n$
- The recursion tree for this recurrence is :



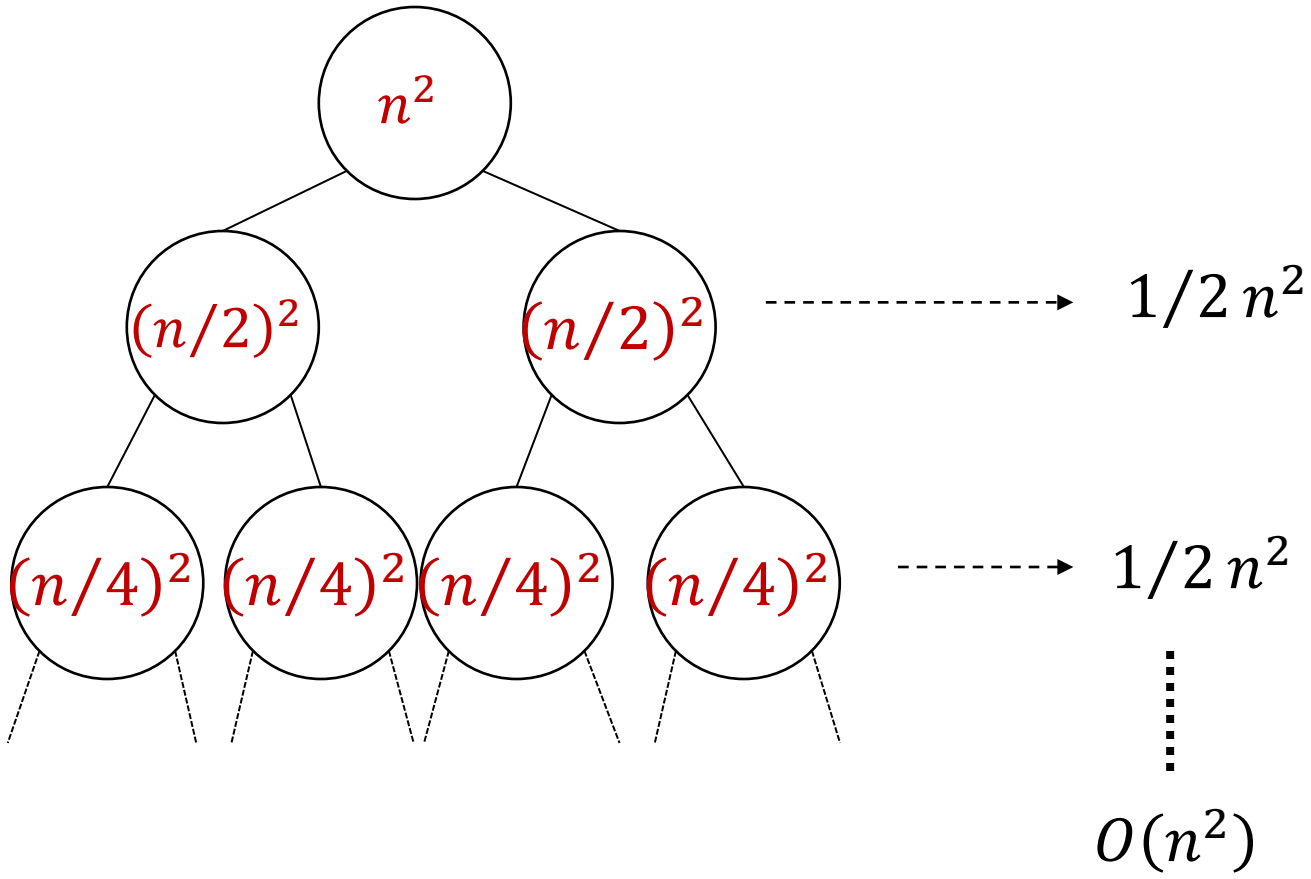
Recurrence Tree Method

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

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

Recurrence Tree Method

- Example 2: $T(n) = 2T(n/2) + c.n^2$
- The recursion tree for this recurrence is :



Recurrence Tree Method

- 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,

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

$$T(n) = 2n^2 + O(n)$$

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

Recurrence Tree Method

- Example 3: $T(n) = T(n/4) + T(3n/4) + c.n$
- Example 4: $T(n) = 3T(n/4) + c.n^2$
- Example 5: $T(n) = T(n/4) + T(n/2) + n^2$
- Example 6: $T(n) = T(n/3) + T(2n/3) + n$

Method of Guessing and Confirming

“guess the answer; and then prove it correct by induction”

What if the given recurrence doesn’t seem to match with any of these (master theorem) methods? If we guess a solution and then try to verify our guess inductively, usually either the proof will succeed (in which case we are done), or the proof will fail (in which case the failure will help us refine our guess).

Example

As an example, consider the recurrence $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$. This doesn't fit into the form required by the Master Theorems. Carefully observing the recurrence gives us the impression that it is similar to the divide and conquer method (dividing the problem into $\sqrt{n}$ subproblems each with size $\sqrt{n}$). As we can see, the size of the subproblems at the first level of recursion is n. So, let us guess that $T(n) = O(nlogn)$, and then try to prove that our guess is correct.

Let's start by trying to prove an upper bound T(n) < cnlogn:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.\, c\sqrt{n}\, log\sqrt{n} + n \\ &= n.\, c\, log\sqrt{n} + n \\ &= n.c.\frac{1}{2}.logn + n \\ &\leq cnlogn \end{aligned}$$

The last inequality assumes only that 1 ≤ c. .logn. This is correct if n is sufficiently large and for any constant c, no matter how small. From the above proof, we can see that our guess is correct for the upper bound.

Change of variable-Method

- There are many ways to solve a recurrence relation runtime. One way to do this is a method called “change of variable”.
- Domain transformations can sometimes be used to substitute a function for the argument of the relation and make it easier to solve.
- The idea is to select a function for S(m), when given T(n).

Example

$n=2^m$ --> $T(2^m)= S(m)$

$m=\log n$

$2^m=2^{\log_2 n}$

$2^m=n$

$2^{m/2}=2^{(\log_2 n/2)}$

$2^{m/2}=n^{1/2}$

$T(n)=2T\sqrt{n}+\log n$

Let's rewrite the equation by substituting $m=\log n$, and then plug it back in to the recurrence to get the following:

$$T(2^m)=2T(2^{m/2})+m$$

Now we will create a new function called 'S' that takes in a parameter 'm' such that $S(m) = T(2^m)$.

Which means : $S(m) = 2T(2^{m/2})+m$.

If $S(m) = T(2^m)$ then $S(m-1) = T(2^{(m-1)})$ and $S(m/2) = T(2^{(m/2)})$, so we can rewrite our function to get the following:

$$S(m)=2S(m/2)+m$$

Using master theorem we can write the solution of recurrence 'S': $O(m\log m)$.

This means that our original function 'T' belongs to $O(\log n * \log(\log n))$, since we can simply replace 'm' with 'log n' because $m=\log n$

Example

$$\mathbf{T(n)=\sqrt{n}T(\sqrt{n})+n}$$

=>First substitute $n = 2^m$

$$T(2^m)=2^{m/2}T(2^{m/2})+2^m$$

Now divide both side by 2^m

$$\frac{T(2^m)}{2^m}=\frac{2^{m/2}T(2^{m/2})}{2^m}+\frac{2^m}{2^m}$$

$$\frac{T(2^m)}{2^m}=\frac{2^{m/2}T(2^{m/2})}{2^m}+1 \qquad \{\frac{T(2^m)}{2^m}=\frac{T(2^{m/2})}{2^{m/2}}+1\}$$

=>second substitution $S(m)=\frac{T(2^m)}{2^m}$

$$S(m)=S(m/2)+1$$

By applying masters method we get : $S(m)= \log m$

Now back substitution => $\frac{T(2^m)}{2^m}=\log m$

$$T(2^m)=2^m \log m$$

Again back-substitute: $T(n)=n \log \log n$

- **$T(n)=3T(n/2)+n$**

Substitute $n=2^m$

$T(2^m)=3T(2^m/2)+2^m$

$T(2^m)=3T(2^{m-1})+2^m$

Substitute $S(m)=T(2^m)$

$S(m)=3S(m-1)+2^m$

$S(m)-3S(m-1)=2^m$

By characteristic equation

$(X-3)(X-2)$

Solution of $S(m)=c_1 3^m+c_2 2^m$

By back substitution

$T(n)=c_1 3^{\log n}+c_2 2^{\log n}$

$=c_1 n^{\log 3}+c_2 n^{\log 2}$

Range transformation

- When we make a change of variable, we transform the domain of the recurrence.
- Instead, it may be useful to transform the range to obtain a recurrence in a form that we know how to solve. Both transformations can sometimes be used together.

Consider the following recurrence, which defines T(n) when n is a power of 2.

$$T(n)=nT^2(n/2)$$

The first step is a change of variable: $n=2^m$, $S(m)=T(2^m)$

$$T(2^m)= 2^m T^2(2^m/2) \Rightarrow T^2(2^{m-1})$$

$$S(m)=2^m S^2(m-1)$$

- At first glance, none of the techniques we have seen applies to this recurrence since it is not linear; To transform the range, we create yet another recurrence by using U(m) to denote log S(m).

$S(m)=2^m S^2(m-1)$

Substitute $U(m)=\log(S(m))$

$=\log(2^m S^2(m-1))$

$=\log 2^m + \log S^2(m-1)$

$=m+2\log S(m-1)$

$=m+2U(m-1)$

$U(m)= m+2U(m-1)$

$U(m)-2U(m-1)=m$

Solve this by applying Characteristic equation:

Divide & Conquer (D&C) Technique

- 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 $t(n)$** taken by this divide-and-conquer algorithm is given by recurrence equation,

$$t(n) = a * t(n/b) + \mathrm{g}(n)$$

- The solution of equation is given as,

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

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

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

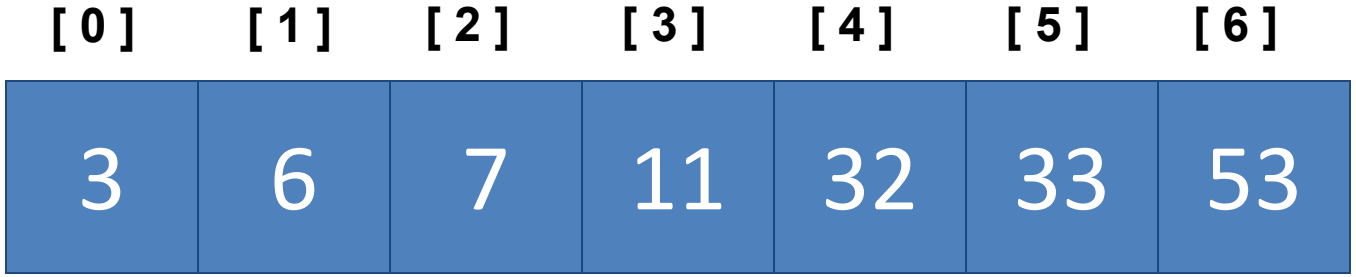
Binary Search

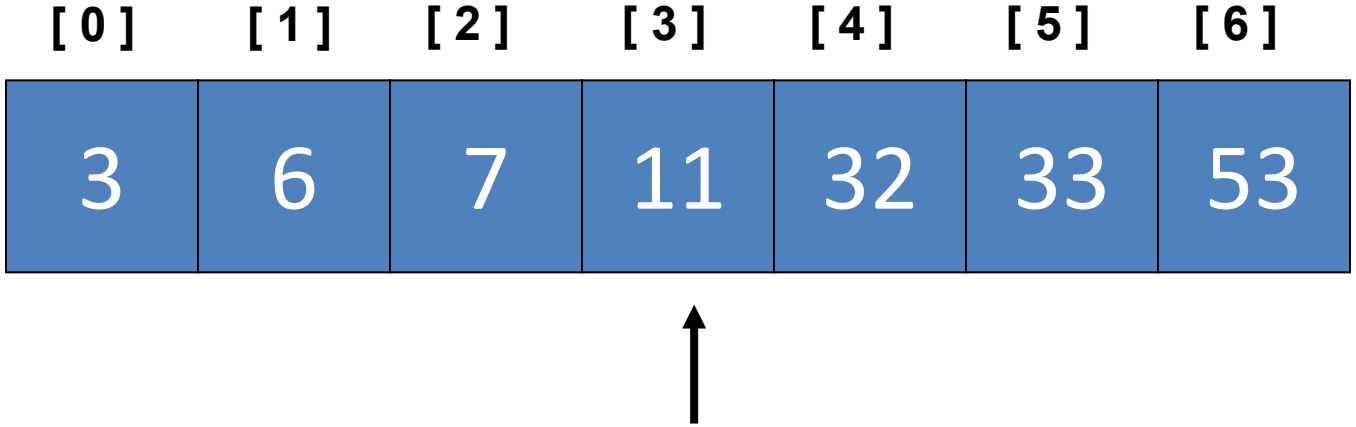
Binary Search

- 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 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.

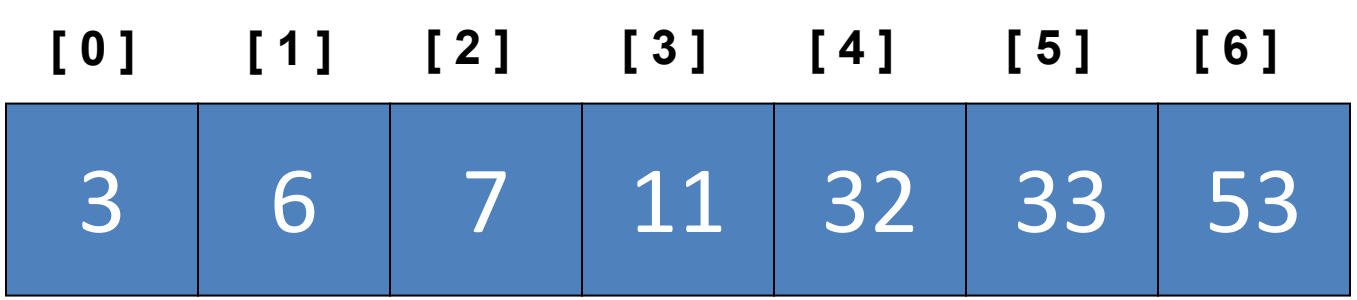
Binary Search - Example

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



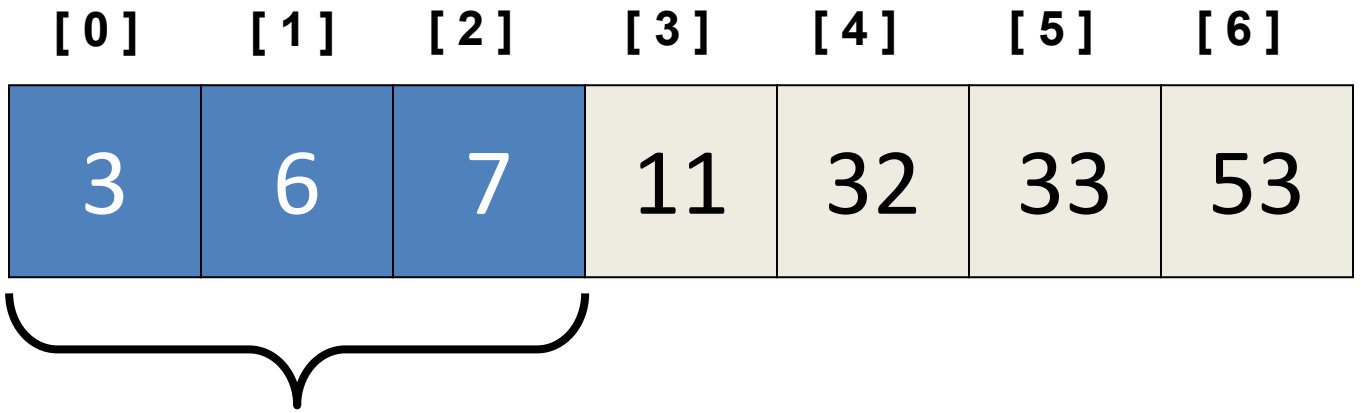


Binary Search - Example



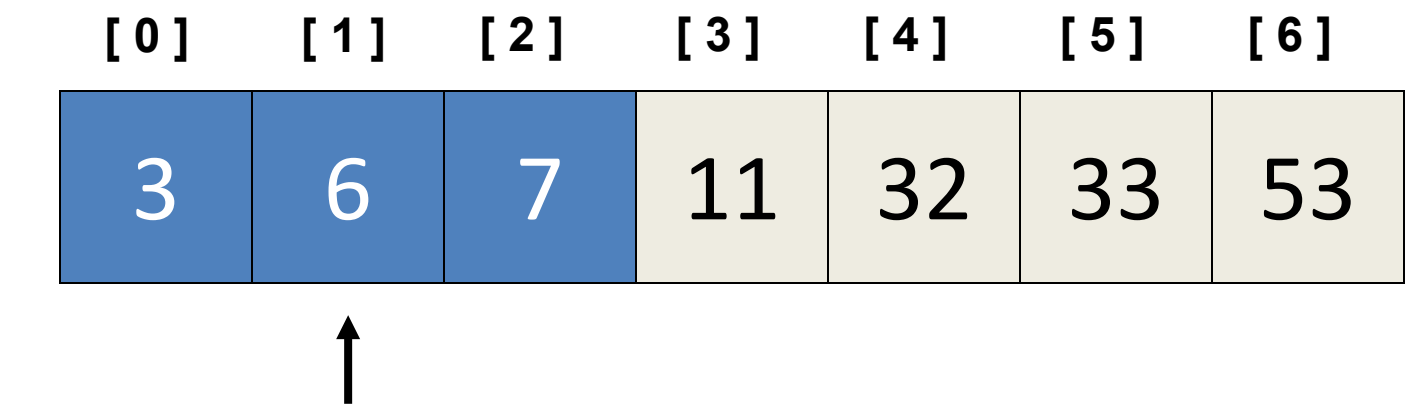
$x = 7$

Is 7 < midpoint value? YES.



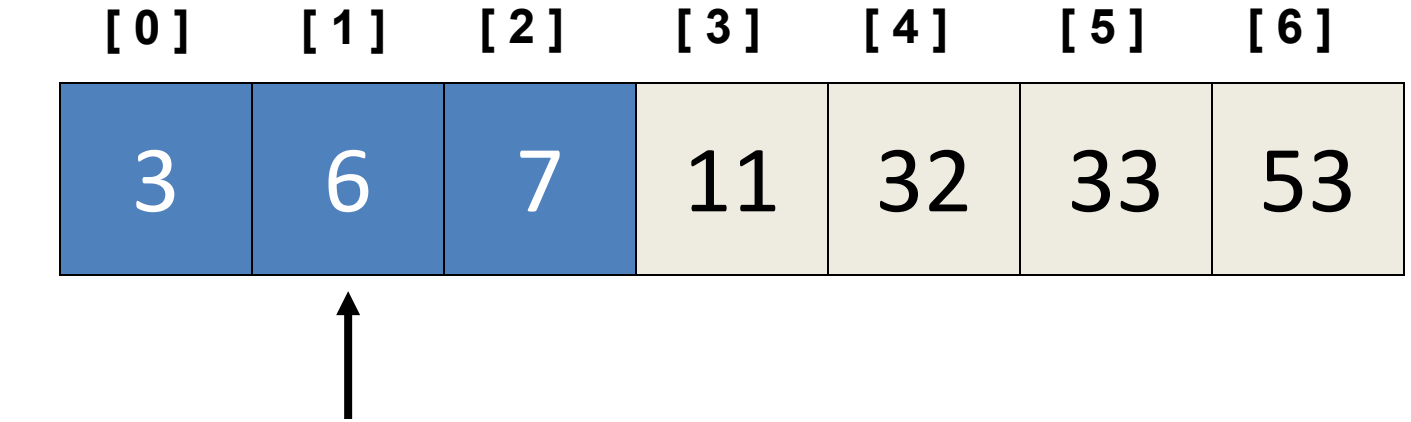
Search for the target in the area before midpoint.

Binary Search - Example



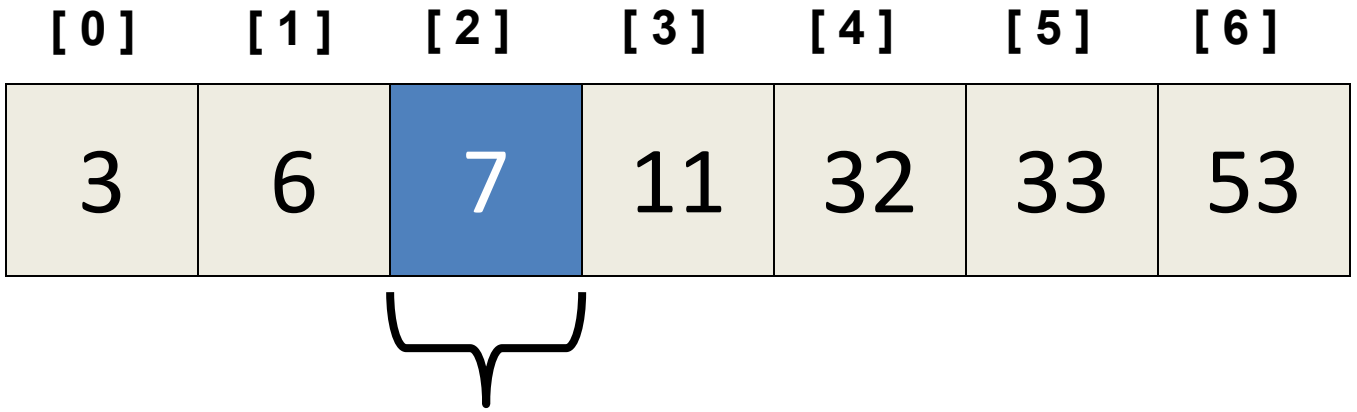
$x = 7$

Find approximate midpoint



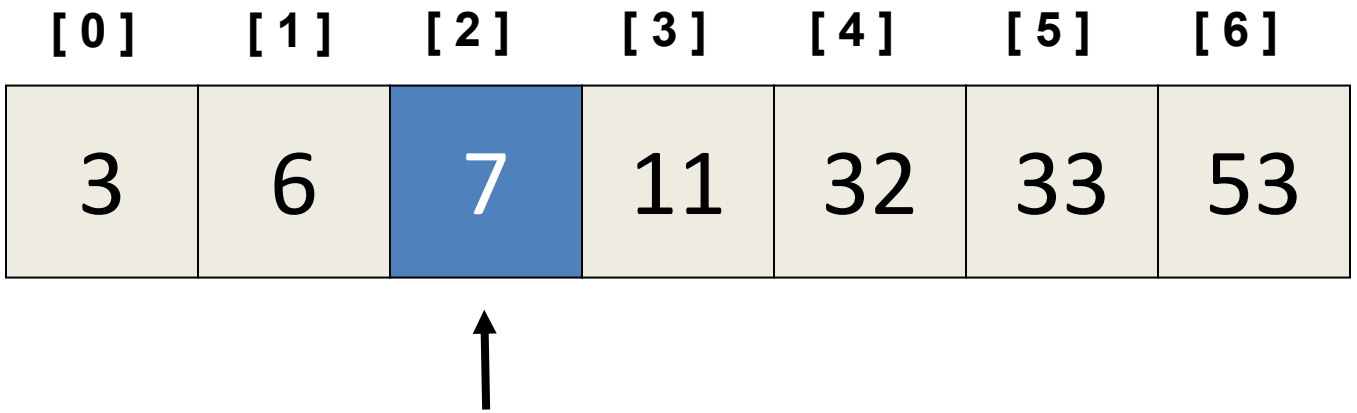
x > value of midpoint? YES.

Binary Search - Example



$x = 7$

Search for the x in the area after midpoint.



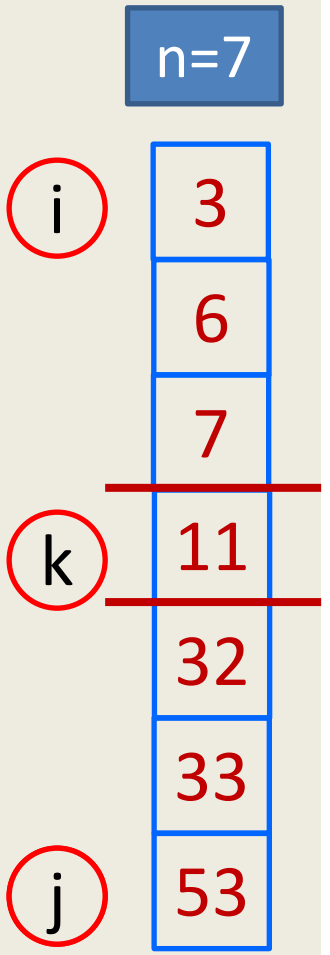
Find approximate midpoint.

Is x = midpoint value? YES.

Binary Search – Iterative Algorithm

- **Algorithm: BinarySearch(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 – Recursive Algorithm

```
BinarySearch(arr, item, beg, end)
        if beg<=end
                midIndex = (beg + end) / 2
                if item == arr[midIndex]
                    return midIndex
                else if item < arr[midIndex]
                    return binarySearch(arr, item, midIndex + 1, end)
                else
                    return binarySearch(arr, item, beg, midIndex – 1)
        return -1
```

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,

$$\boldsymbol{t(n) = t(n/2) + \theta(1)}$$

- Comparing this to the general template for divide and conquer algorithm, $a = 1, b = 2 \; and \; c = 0$.

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

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

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

Binary Search – Examples

1. Demonstrate binary search algorithm and find the element $x = 12$ 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 $x = 31$ in the following array. [7]

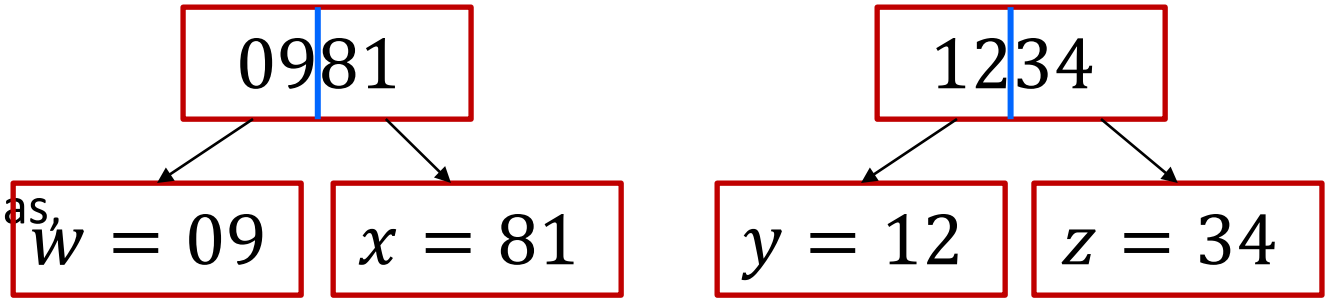
$$10, 15, 18, 26, 27, 31, 38, 45, 59$$

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

Multiplying Large Integers

Multiplying Large Integers Problem

- 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:



  2. We can write as,

$$0981 = 10^2 w + x$$

$$1234 = 10^2 y + z$$

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

Multiplying Large Integers Problem

- Now, the required product can be computed as,

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

$$0981 \times 1234 = (10^2 w + x) \times (10^2 y + z)$$
$$= 10^4 w \cdot y + 10^2 (w \cdot z + x \cdot y) + x \cdot z$$
$$= 1080000 + 127800 + 2754$$
$$= 1210554$$

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

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

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

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

- Only one multiplication is required instead of two.

Additional terms

Multiplying Large Integers Problem

- Now we can compute the required product as follows:

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

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

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

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

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

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

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

$$= 1080000 + 127800 + 2754$$

$$= 1210554.$$

for Binary Number

- a= | a_L | a_R |
- b= | b_L | b_R |

$$w = a_L$$
$$x = a_R$$
$$y = b_L$$
$$z = b_R$$

$$p = w \cdot y = a_L * bL$$

$$q = x \cdot z = a_R * bR$$

$$r = (w + x) \times (y + z) = (a_L + a_R) \times (b_L + b_R)$$

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

$$a \cdot b = 2^n a_L \cdot b_L + 2^{n/2}(a_L \cdot b_R + a_R \cdot b_L) + a_R \cdot b_R$$

$$a \times b = 2^n p + 2^{n/2}(r - p - q) + q$$

Algorithm

Multiply(a, b, n)

if n = 1 return a*b

a_L= The left half of a

a_R= The right half of a

b_L= The left half of b

b_R= The right half of b

p= Multiply(a_L, a_R, n/2) //recursive call

q= Multiply(b_L, b_R, n/2) //recursive call

r= $((a_L + a_R)\ (b_L + b_R)$, n/2) //recursive call

return $2^n p + 2^{n/2}(r - p - q) + q$ //

time taken by p, q, and r are n/2, while $2^n p$ appending n zeros to right will be proportional to n, addition and subtraction are proportional to n.

Analysis of Time Complexity

- 981×1234 can be reduced to **three multiplications** of two-figure numbers $(09 \cdot 12, 81 \cdot 34\ and\ 90 \cdot 46)$ 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,

$$\boldsymbol{T(n) = 3t\,(n/2) + g(n)},\ \text{vs}\ \boldsymbol{T(n) = 4t\,(n/2) + g(n)},$$

Solving it gives,

$$\boldsymbol{T(n) \in \theta\left(n^{log3}\right) = \mathbf{n}^{1.59}}$$

Multiplying Large Integers Problem

- 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

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

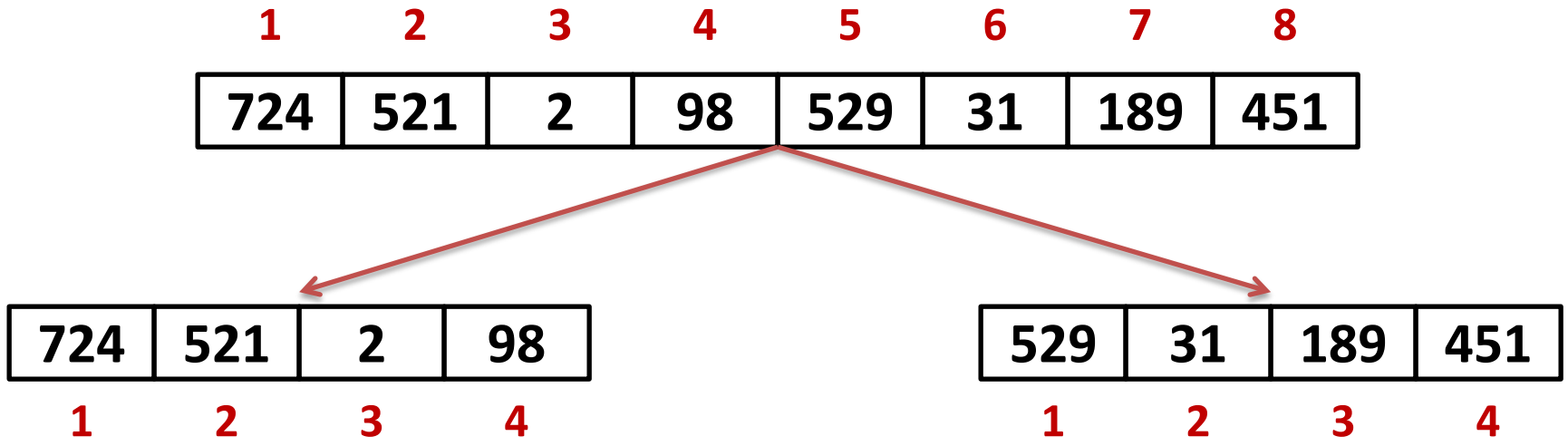
Merge Sort

Merge Sort - Example

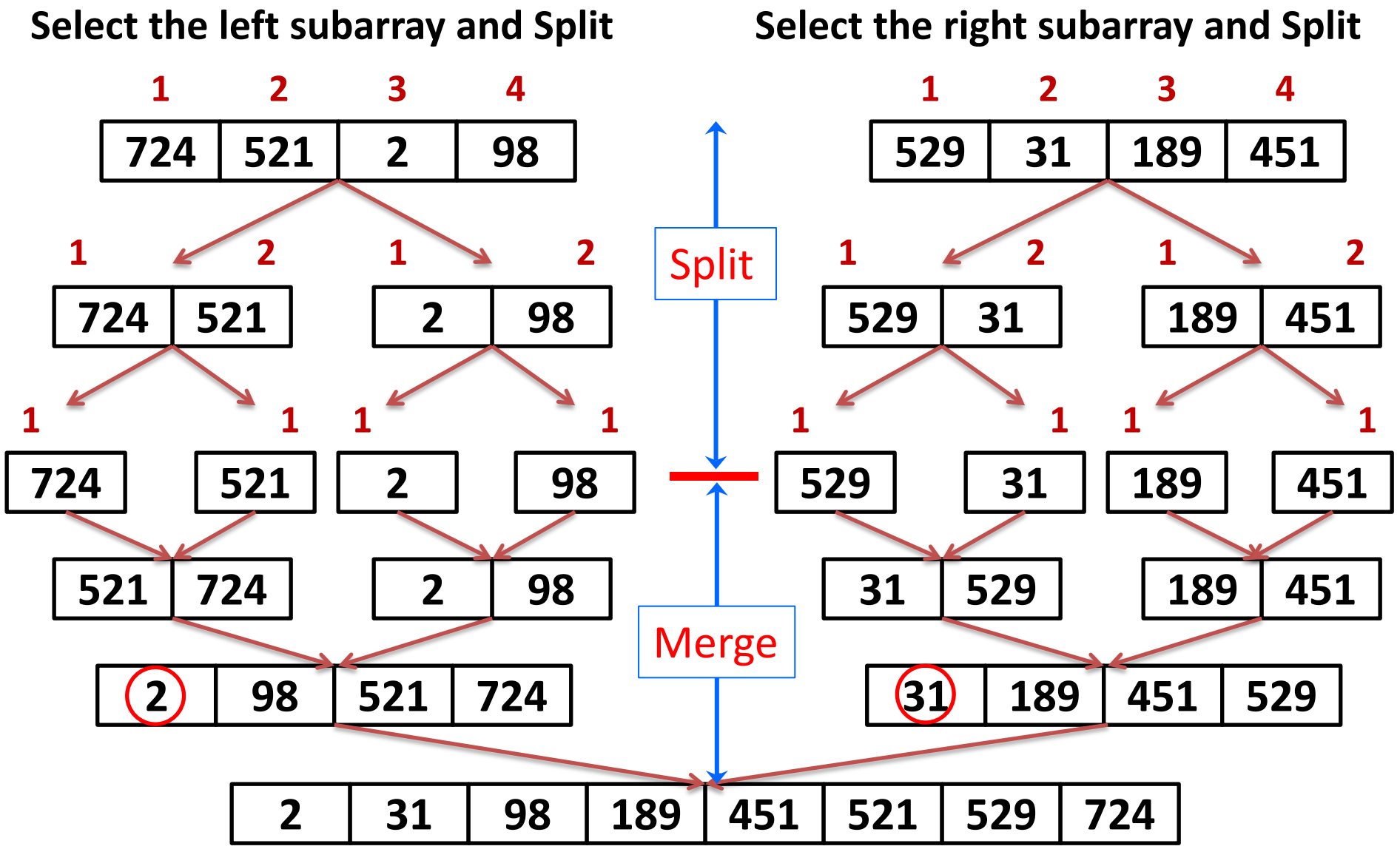
Unsorted Array

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

Step 1: Split the selected array



Merge Sort - Example



Merge Sort – Example (HW)

- Sort given numbers into descending order using merge sort.

38, 27, 43, 3, 9, 82, 10, 67, 71, 54, 97

Merge Sort - Algorithm

```
MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)
```

```
merge(U[p..q],V[q+1..r],T[])
        i ← 1;
        j ← 1;
        U[q+1], V[r+1] ← ∞;
        for k ← 1 to q + r do
            if U[i] < V[j]
                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**.

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

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

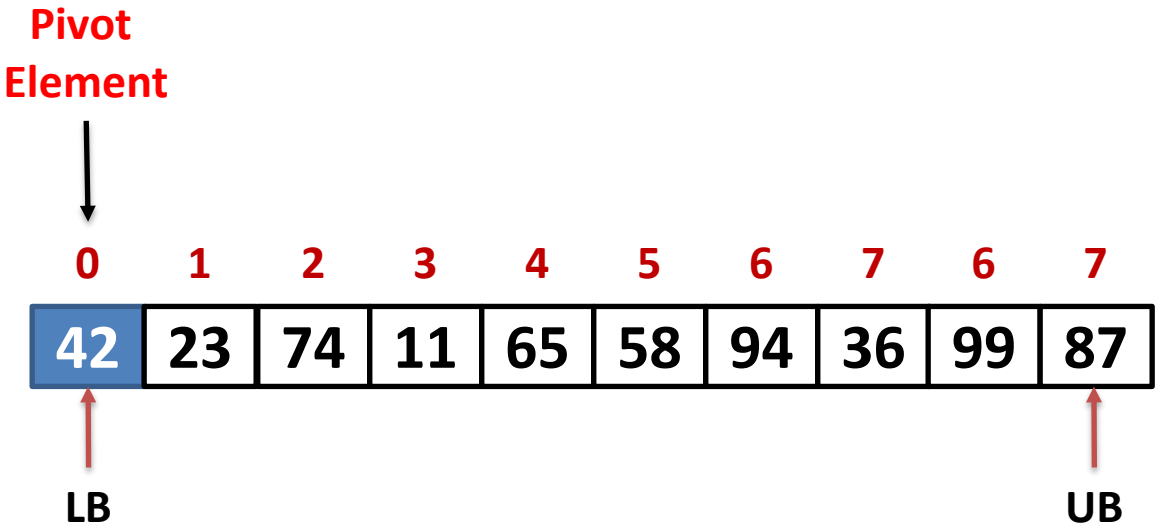
- Applying the general case, $a =$ $t(n) = at(n/b) + g(n)$
- Since $a = b^k$ the **second case** applies so, $t(n) \in \theta(nlogn)$.
- Time complexity of merge sort is $\boldsymbol{\theta(nlogn)}$.

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

Quick Sort

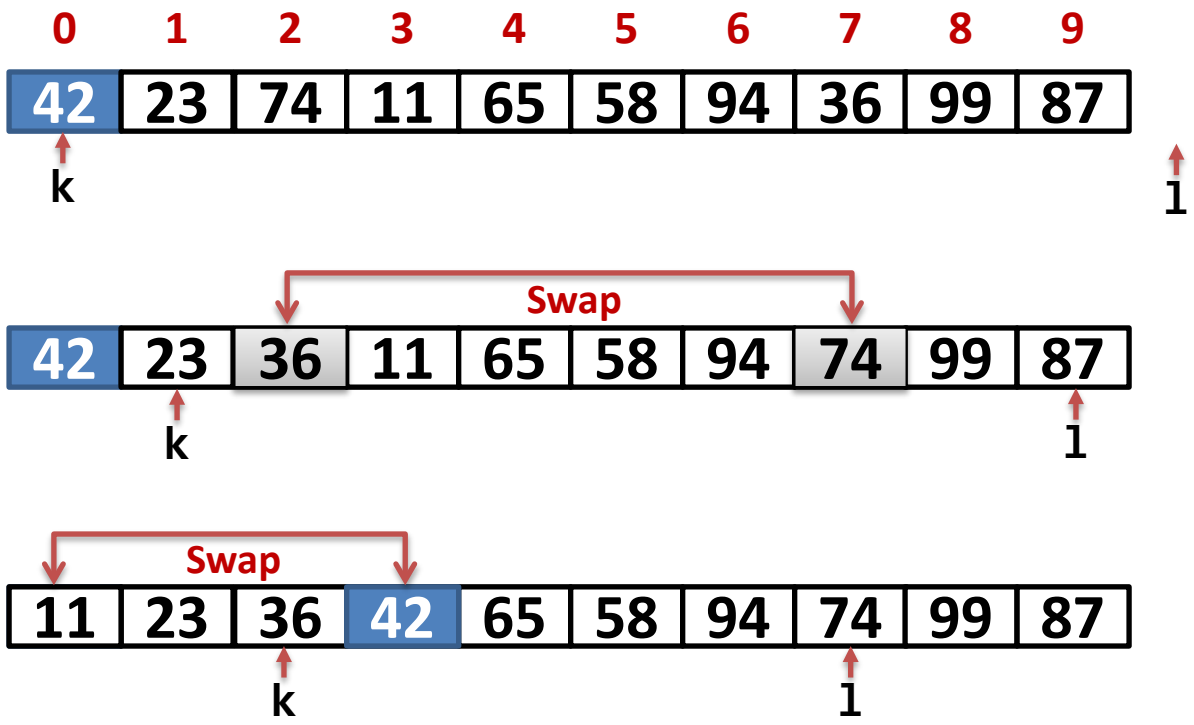
Quick Sort – Example(Inplace)

- 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 - 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]
```



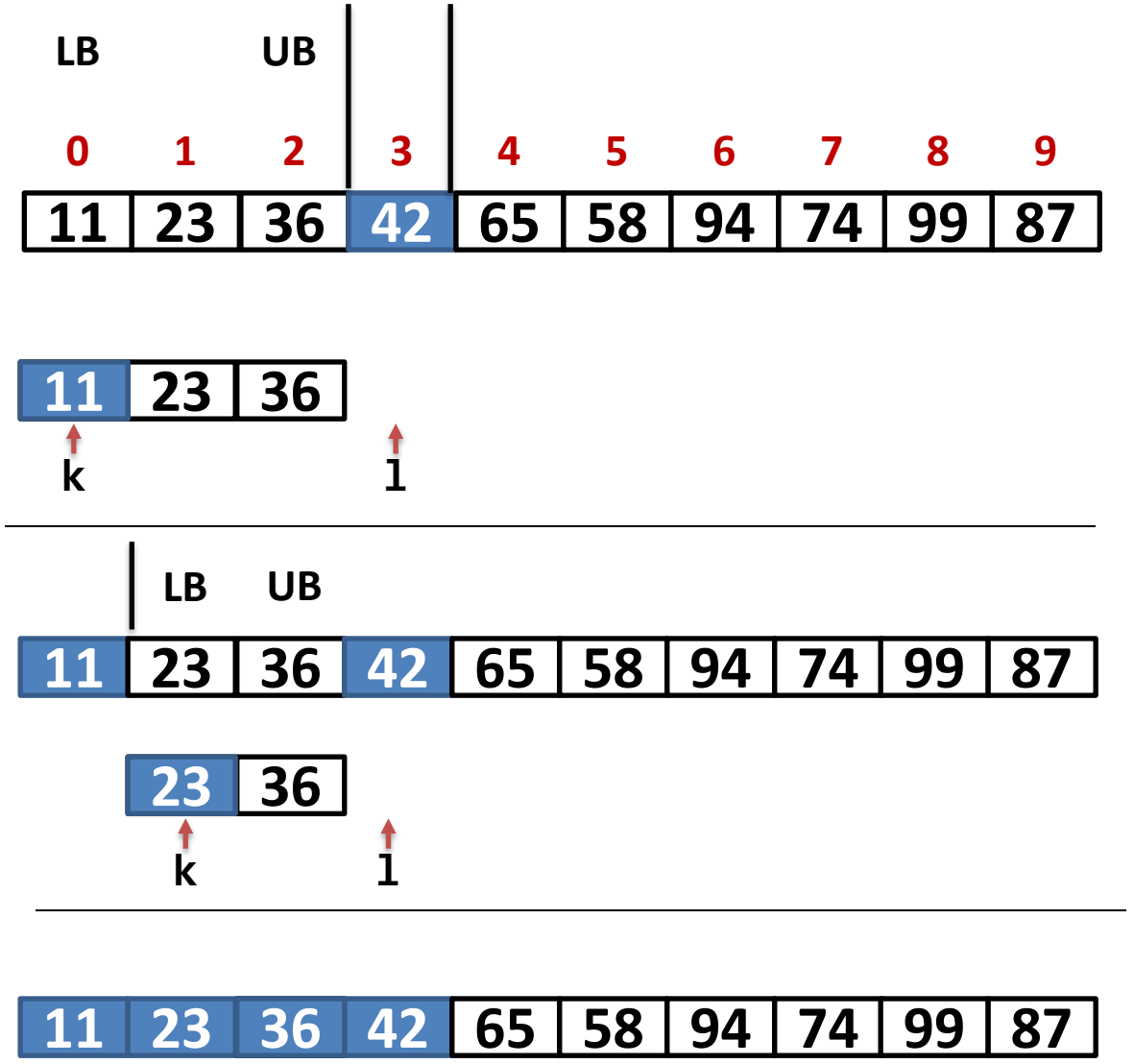
LB = 0, UB = 9 p = 42

k = 0

l = 10

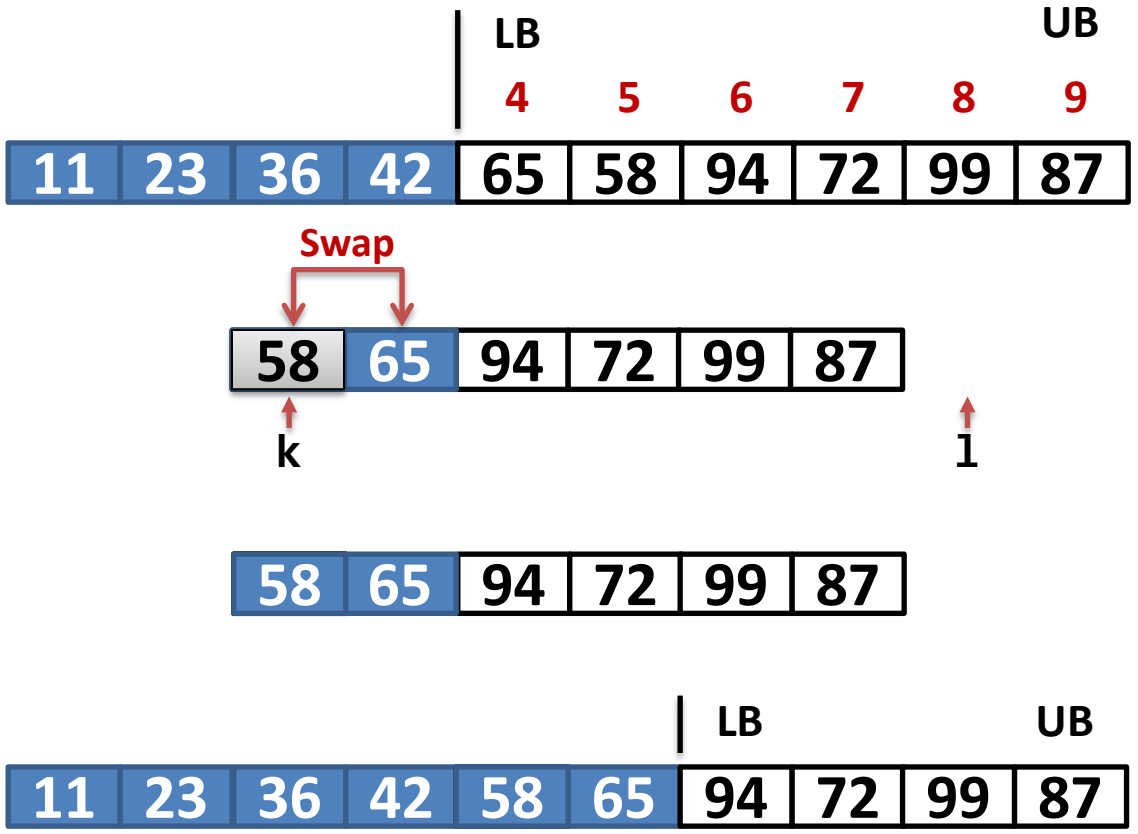
Quick Sort - Example

```
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]
```



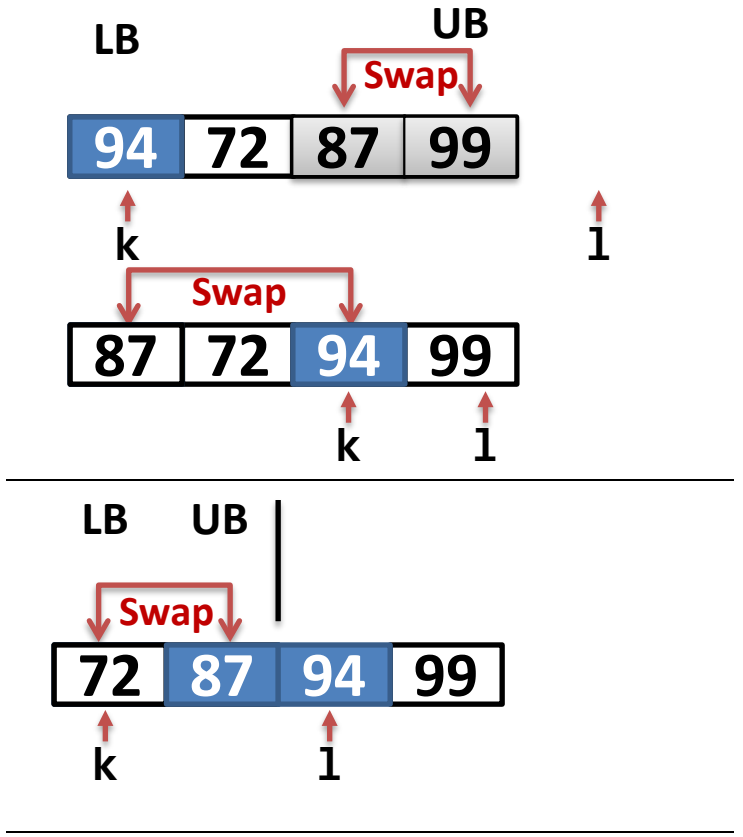
Quick Sort - Example

```
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 - 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]
```



11	23	36	42	58	65	72	87	94	99

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 (HW)
  3. 9, 7, 5, 11, 12, 2, 14, 3, 10, 6 (HW)

Quick Sort - Algorithm

```
Partition(A,p,r)
01  x←A[r]
02  i←p-1
03  j←r+1
04  while TRUE
05      repeat j←j-1
06          until A[j] ≤x
07      repeat i←i+1
08          until A[i] ≥x
09      if i<j
10          then exchange A[i]↔A[j]
11          else return j
```

```
Quicksort(A,p,r)
01  if p<r
02      then q ←Partition(A,p,r)
03           Quicksort(A,p,q)
04           Quicksort(A,q+1,r)
```

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,

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

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

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

- $\theta(n)$: $cost\ of\ partitioning\ at\ each\ level$, which is i + j times we are moving, in this case $\mathrm{i} = \mathrm{n} - 1$ and $\mathrm{j} = 1$
- When does the worst case appear?: when the element are sorted.

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,

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

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

Strassen's Algorithm for Matrix Multiplication

Matrix Multiplication

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

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

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

- 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.

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

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

$$C_{i,j} = \sum_{k=1}^{n} (A_{ij} * Bkj)$$

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

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

Algorithm for simple Addition

```
Addition(A,B) //A,B,C are two dimensional matrices.
N ←  rows[A]
Let C be the matrix having A+B of size n*n.
for i← 0 to n
    for j← 0 to n
          C[i][j] ← (A[i][k]+B[k][j])
return C
```

Time Complexity of above method is $O(n^2)$.

Algorithm for simple multiplication

```
Naive_square_multiply(A,B) //A,B,C are two dimensional matrices.
n ←  rows[A]
Let C be the matrix having A*B of size n*n.
for i← 0 to n
    for j← 0 to n
        for k← 0 to n
             C[i][j] ← C[i][j] +(A[i][k]*B[k][j])
return C
```

Time Complexity of above method is $O(n^3)$.

Divide and Conquer Approach

Let us first assume that n is an **exact power of 2** in each of the **n x n** matrices for A and B. This simplifying assumption allows us to break a big **n x n** matrix into smaller blocks or quadrants of size **n/2 x n/2**, while also ensuring that the dimension **n/2** is an integer.

- We calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

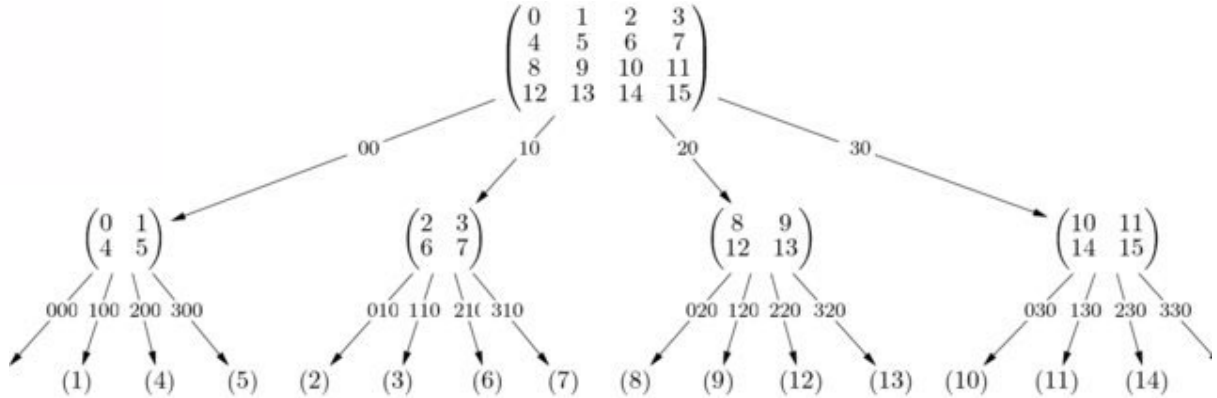
A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2



In the above method, we do **8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time**. So the time complexity can be written as

$$T(N) = 8T(N/2) + O(N^2)$$

$$O(N^3)(\text{no better than naïve algorithm})$$

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

```
1  n = A.rows
2  let C be a new n × n matrix
3  if n == 1
4      c11 = a11 · b11
5  else partition A, B, and C as in equations (4.9)
6      C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
7      C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
8      C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
9      C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
           + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10 return C
```

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

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

Logic

- For multiplying two matrices of size n x n, we make 8 recursive calls above, each on a matrix/subproblem with size n/2 x n/2. Each of these recursive calls multiplies two n/2 x n/2 matrices, which are then added together. For the addition, we add two matrices of size $n^2/4$, so each addition takes $\Theta(n^2/4)$
- time. We can write this recurrence in the form of the following equations

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

Hence, recurrence **$T(N) = 8T(N/2) + O(N^2)$** **i.e $O(N^3)$**

MOTIVATION BEHIND STRASSEN ALGORITHM

Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?

- In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of Strassen's method is to reduce the number of recursive calls to 7. Multiplication is more costly operation than addition.
- Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

$$\mathbf{T(N) = 7T(N/2) + O(N^2)}$$

- From Master's Theorem, time complexity of above method is $O(N^{\log_2 7})$ which is approximately $O(N^{2.8074})$ i.e $O(N^{2.81})$

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

p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

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**.

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

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

- The general equation applies with $a = 7, b = 2$ and $k = 2$.
- Since $a > b^k$, the **third case** applies and $t(n) \in O\left(n^{lg7}\right)$.
- Since $log_2 7 > 2.81$, it is possible to multiply two $n \times n$ matrices in a time $\boldsymbol{O(n^{2.81})}$.

Exponentiation

Exponentiation - Sequential

- Let a and n be two integers. We wish to compute the **exponentiation** $x = a^n$.
- 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 – D & C

- The efficient exponentiation algorithm is based on the simple observation that for an even n, $a^n=a^{(n/2)} * a^{(n/2)}$.
- The case of odd b is trivial, as it's obvious that $a^n=a * a^{(n-1)}$.
- So now we can compute by doing only log(n) squarings and no more than log(n) multiplications, instead of n multiplications - and this is a very improvement for a large n.
- **Suppose, we want to compute a^{10}**

We can write as,

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

- In general,

$$a^n = \begin{cases} 1 & if\ n = 0 \\ a & if\ n = 1 \\ \left(a^{n/2}\right)^2 & if\ n\ is\ even \\ a \times a^{n-1} & otherwise \end{cases}$$

Exponentiation – D & C

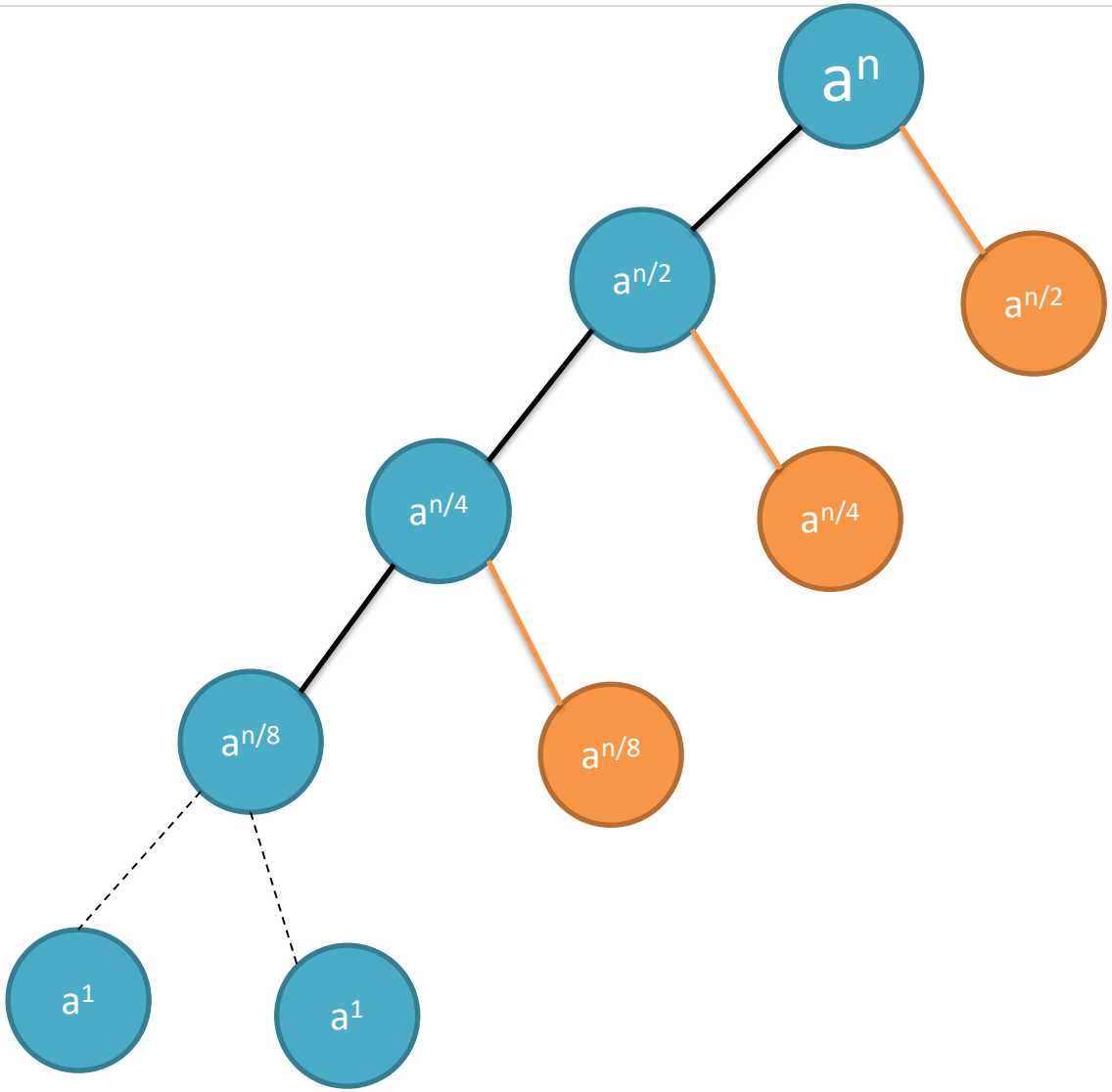
- Algorithm using **Divide & Conquer Approach:**

```
function expoDC(a, n)
  if n = 0 then return  1
  if n = 1 then return  a
  if n is even then return [expoDC(a, n/2)]²
  else return a * expoDC(a, n - 1)
```

- Number of operations performed by the algorithm is given by,

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

Exponentiation – D & C



- $N/2^x = 1$
- $N = 2^x$
- $\log N = x$
- $O(\log N)$
- $N=2^{12}$
- Speed of the machine is 2^{10} / second

Thank You!