RNS
ESTD 2001
An Institute with a Difference
RNSIT-CSE
Bengaluru
Founder
Dr. R N Shetty
Vision - Preparing Better Computer Professionals for a Real World

Module-1

Design and Analysis of Algorithms

Introduction

Contents

- **Introduction:**
  What is an Algorithm? (T2:1.1), Algorithm Specification (T2:1.2), Analysis Framework (T1:2.1), Performance Analysis: Space complexity, Time complexity (T2:1.3).

- **Asymptotic Notations:**

Big-Oh notation (O), Omega notation (Ω), Theta notation (Θ), and Little-oh notation (o), Mathematical analysis of Non-Recursive and recursive Algorithms with Examples (T1:2.2, 2.3, 2.4). Important Problem Types: Sorting, Searching, String processing, Graph Problems, Combinatorial Problems.

- **Fundamental Data Structures:**
  Stacks, Queues, Graphs, Trees, Sets and Dictionaries. (T1:1.3,1.4).

Who Coined the word Algorithm?

- Mohammad ben Musā Khwārazmi(780 – 850), Arabized as al-Khwarizmi and formerly Latinized as Algorithmi, was a Persian polymath who produced vastly influential works in mathematics, astronomy, and geography.

- In Europe, the word "algorithm" was originally used to refer to the sets of rules and techniques used by Al-Khwarizmi to solve algebraic equations, before later being generalized to refer to any set of rules or techniques.

Why Algorithms?

Airways Route   Xerox Shop

Why Algorithms?

- Is the water safe& cleansified to drink?
- What are the affects of Climate Change?
- To Check documents similarity
- Design the self Driven cars.

1.1 What is an Algorithm? (T2:1.1)

- Abstract Computational procedure which accept valid input and

- **Program= Expression of an Algorithm.**
- **Algorithm are intended**

produce valid output.

to be read by human beings.

Program= Expression of an Algorithm

1.1 What is an Algorithm? (T2:1.1)

- – Step by Step procedure to solve a given problem.
- – Method used by a computer for the solution of the problem.

- An algorithm is a sequence of computational steps that transforms the input into the output.

“An Algorithm is a finite set of instructions that if followed , accomplishes a particular task”.

“An Algorithm is a sequence of unambiguous instructions for solving a problem i,e for obtaining a required output for any legitimate input in a finite amount of time”

Program= Expression of an Algorithm

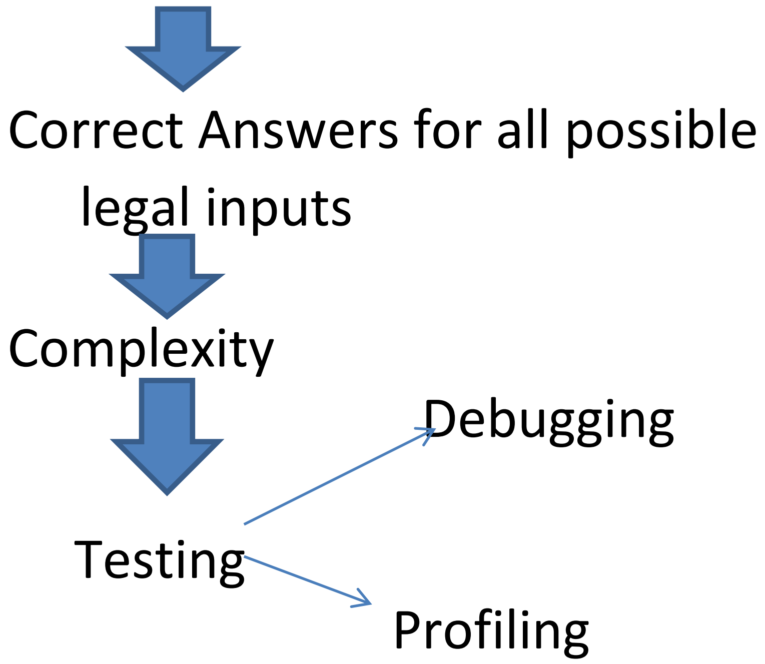
Criteria for Algorithm

- **Input-** Zero or more quantities externally supplied.
- **Output-** At least one quantity is produced.
- **Definiteness** – Each Instruction is clear and unambiguous.
- **Finiteness:** Algorithm terminates at finite number of steps.
- **Effectiveness:** Every instruction can be implemented.

The study of Algorithms include

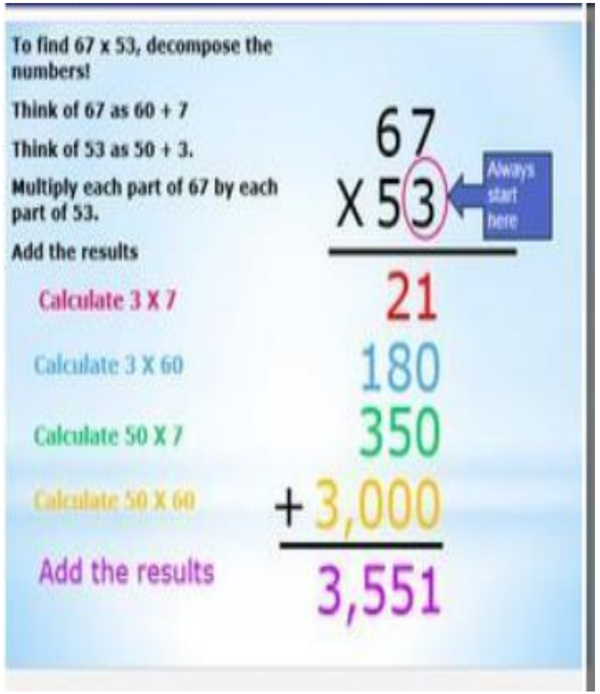
1. How to devise Algorithms?
2. How to validate Algorithms?
3. How to analyze Algorithms?
4. How to test a program?

Various Approaches


Correct Answers for all possible legal inputs
Complexity
Testing
Debugging
Profiling

1.2 Algorithm Specification (T2:1.2)

- **Natural**


To find 67 x 53, decompose the numbers!

Think of 67 as 60 + 7

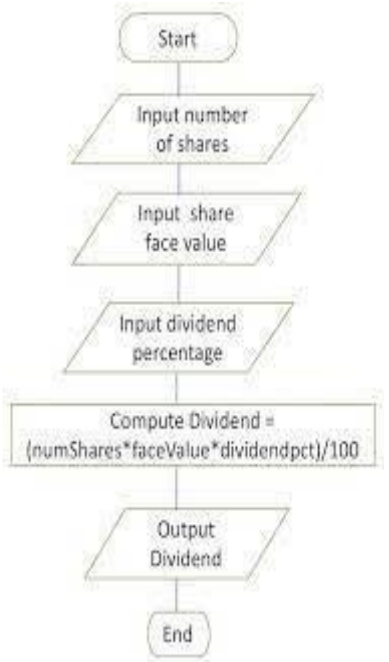
Think of 53 as 50 + 3.

Multiply each part of 67 by each part of 53.

Add the results

```
      67
    X 53
    ----
      21    Calculate 3 X 7
     180    Calculate 3 X 60
     350    Calculate 50 X 7
 + 3,000    Calculate 50 X 60
 -------
   3,551    Add the results
```

Always start here


Start
Input number of shares
Input share face value
Input dividend percentage
Compute Dividend = (numShares*faceValue*dividendpct)/100
Output Dividend
End

RepresentationInstructions must be definite and clear.

- **Graphical Representation**They work only if algorithm is simple and small.

Pseudo Code Convention

- Comments begin with //
- Compound statements/Blocks are specified with a pair of matching braces i,e { ……
  }
  - Statements are delimited by ;

- An identifier/variable begins with letter and data types of variables are not explicitly specified.
- Assignment of values to variables is done using assignment statement
  - Variable:=expression;   variable <- value/exp
- Logical Operators(‘and’, ‘or’ & ‘not’) and Relational Operators(<,<=,>,==,>=) are provided.
- Array elements are accessed using A[i,j].

- Looping

```
while<condition> do
    {
    }
for variable=value1 to value n step step  do
    {
    }
repeat <statement>
      .
```

```
for i= 1 to n do
{

}
```

until<condition>

– break & return are used to exit from a loop and function.

8. Conditional Statements

        if(condition) then <statements>

    Case statements

9. Input/output – read/write

10. Procedure: Algorithm Name(<parameter list>)

Write?

- Driving directions from • Recipe for cooking your your home to college. favorite dish.

```
Algorithm Max(a1,a2)
// finds Maximum of 2 number
{ if(a1>a2) write a1 is greater;
              else write a2  is
greater; }
```

Puzzle

- A peasant find himself on a river bank with a wolf, a goat and a cabbage. He need to transport all the three to the other side of the river in the boat. However , the boat has room for only the peasant himself and one other item(either wolf/goat/cabbage). In his absence , the wolf would eat the goat, and the goat can eat cabbage. Solve the problem for peasant or show it has no solution.

"Not everything that can be counted counts, and not everything that counts can be counted".

Albert Einstein(1879-1955)

Analysis Framework (T1:2.1)

BasicIdea:

Mathematical Model for a computer

- Mentally executed.
- Will evaluate the time.

What is the time taken?
What is Input?

How does model relate to real computer.

Analysis Framework (T1:2.1)

- **Analysis** is the process of breaking a complex topic or substance into smaller parts in order to gain a better understanding of it.

- The process of studying or examining something in an organized way to learn more about it, or a particular study .

- The separation of an intellectual or substantial whole into its constituent parts for individual study.

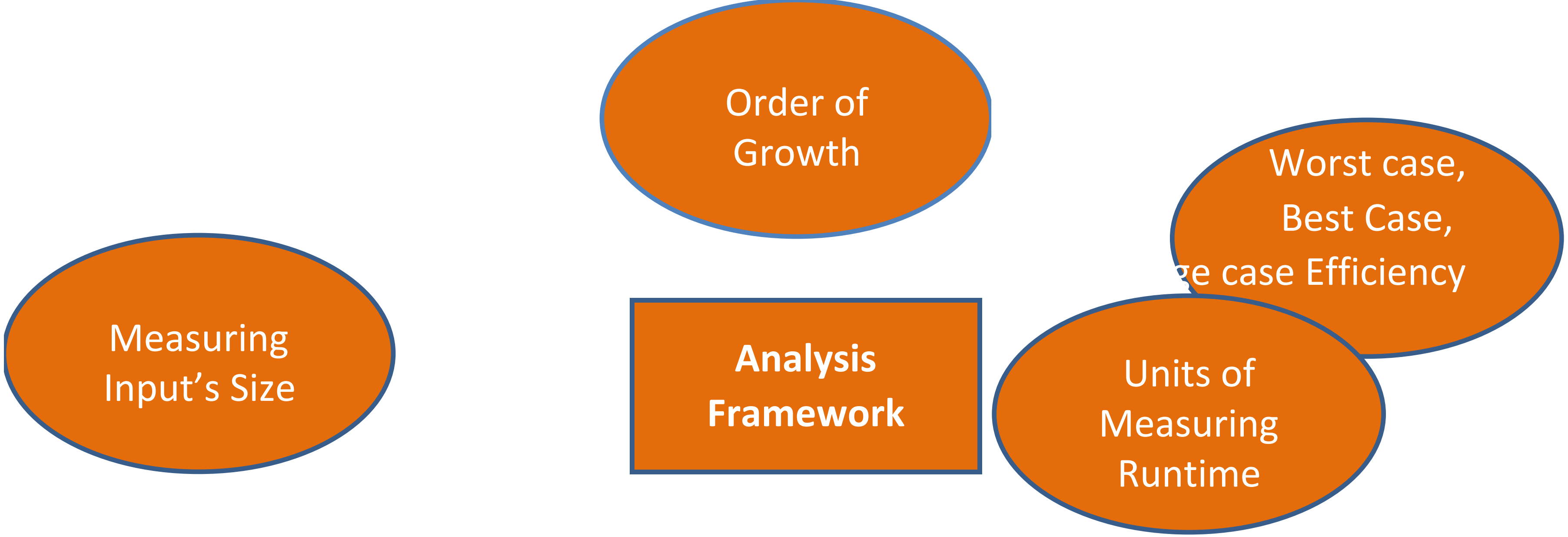
Analysis Framework

- Analysis Framework is • Framework for a systematic approach analyzing the efficiency that can be applied for of algorithms is through analyzing any given – ***Time Efficiency***
  algorithm. – ***Space Efficiency***

***Time efficiency* indicated how fast an algorithm in question runs**

***Space efficiency* deals with extra space/memory units the algorithm requires.**

Factors for Analysis Framework


Order of Growth
Worst case, Best Case, ge case Efficiency
Measuring Input's Size
Analysis Framework
Units of Measuring Runtime

Measuring Input Size

- Almost all algorithms run longer on larger inputs
  - it takes longer to sort larger arrays, multiply larger matrices, and so on

- How about measuring algorithm's efficiency as a function of some parameter *n (in few cases more than one parameters)* indicating the algorithm's input size ?

- Is this selection of parameter n straightforward ?

- Guess the input size for the following problems
  - Sorting
  - Searching
  - Smallest element in a list
  - Matrix multiplications

- Polynomial evaluation – Polynomial addition
- Spell check algorithm

5/28/2021– Primality checking ….

Measuring Input Size

- $b = [\log_2 n] + 1$

b= number of bits.

n=input parameter.

Units of Measuring Runtime

- **Basic Operation**: Most important operation of an algorithm.
- $T(n) = C_{op} C(n)$

Simple programs

```
Algorithm Sum(n)
// finds sum of given numbers
{ s = 0 for i = 1 to n do s = s + i ;
    return s
}
```

Algorithm Max(A[n])

RNSIT-CSE Bengaluru

```
// finds Largest of given Array
{
        max= A[0];
        for i=1 to n-1 do
        { if (max< A[i]) then
                {
                        max= A[i];
                }
        }
}
```

- Assume $C(n) = 1/2\, n(n-1)$, how much longer will the algorithm run if we double its input size?

$C(n) = \frac{1}{2} n(n-1) = \frac{1}{2} n2 - \frac{1}{2} n === \frac{1}{2} n2$

$T(n) = Cop\ C(n)$

$T1(n)/ = \frac{1}{2} n2$

$T2(2n) = \frac{1}{2} 4n2$

Order of Growth

	n			
	10	50	100	1000
Log n	0.00003sec	0.00005 sec	0.00007 sec	0.00009 sec
n1/2	0.00003sec	0.00007 sec	0.000010 sec	0.00032 sec
n	0.00010sec	0.00050 sec	0.00100 sec	0.01000sec
n log n	0.00033sec	0.00282 sec	0.00664 sec	0.09966sec

n2	sec	sec	min	hrs
n3	sec	min	hrs	day
n4	min	day	yrs	cen
2n	sec	yrs	Cen	cen
n!	362.88 sec	1*10 51 cen		

Efficiencies

- Worst Case   While i<n and A[i] = k do

- Best Case i=i+1;
- Average Case If i<n return i

  else return -1

1.4 Performance Analysis (T2:1.3)

Performance Analysis of an algorithm depends upon two factors :

- **Time Complexity**
- **Space Complexity**

– Amount of Computer – Amount of memory it time it needs to run to needs to run to completion completion.

Space Complexity

- Space needed by each of algorithms is the sum of the following components:
  - Fixed Part
  - Variable Part
- ***<u>Fixed Part</u>*** : Independent of characteristics of input and output. It includes Instruction

Space Complexity

space, space for simple variables, etc..

- ***Variable Part***: Dependent on particular problem instance, referenced variables, recursion stack

- Space requirement
- $S(P) = c + S_p$
- S_p: Instance Characteristics/variable Part
- c: Constant/Fixed part.

- When analyzing the space complexity of an algorithm , Sp is of more concern

Simple Examples

```
{
    int z = a + b + c;
    return(z);
```

$S(P)= c+ S_p$

$S(P)= 12+0$ bytes

```
}
{   • N+3 s = 0;
    for  i = 1 to n do
    { s= s + a[i];
    }
    return s;
}
```

```
Algorithm Rsum(a,n)
{   3(n+1) if(n<=0) then return 0;
    else return Rsum(a,n-
          1)+a[n]
}
```

- S(Rsum (a,n))=

Puzzle

- Glove Selection: There are 22 gloves in a drawer.5 pairs of red gloves , 4 pairs of yellow and 2 pairs of green .You select gloves in dark and can check them only after selection of all. What is the smallest number of gloves you need to select to have at least one matching pair in the best case? Worst case?

Time Complexity

- T(P)= Compile Time+ Run Time.
- Runtime=t_P
- Identify program Step

- Program step is loosely defined as a syntactically or semantically a meaningful statement which is independent of instance characteristics.

- For iterative step counts must be considered.

```
{
    S=0; for i=1 to
    n do
            s=s+a[i];
    return s
}
```

EXAMPLES

```
Algorithm Rsum(a,n)
{
```

- 2 + t Rsum(n-1)

if(n<=0) then return 0; • 2+2+ t Rsum(n-2) else .

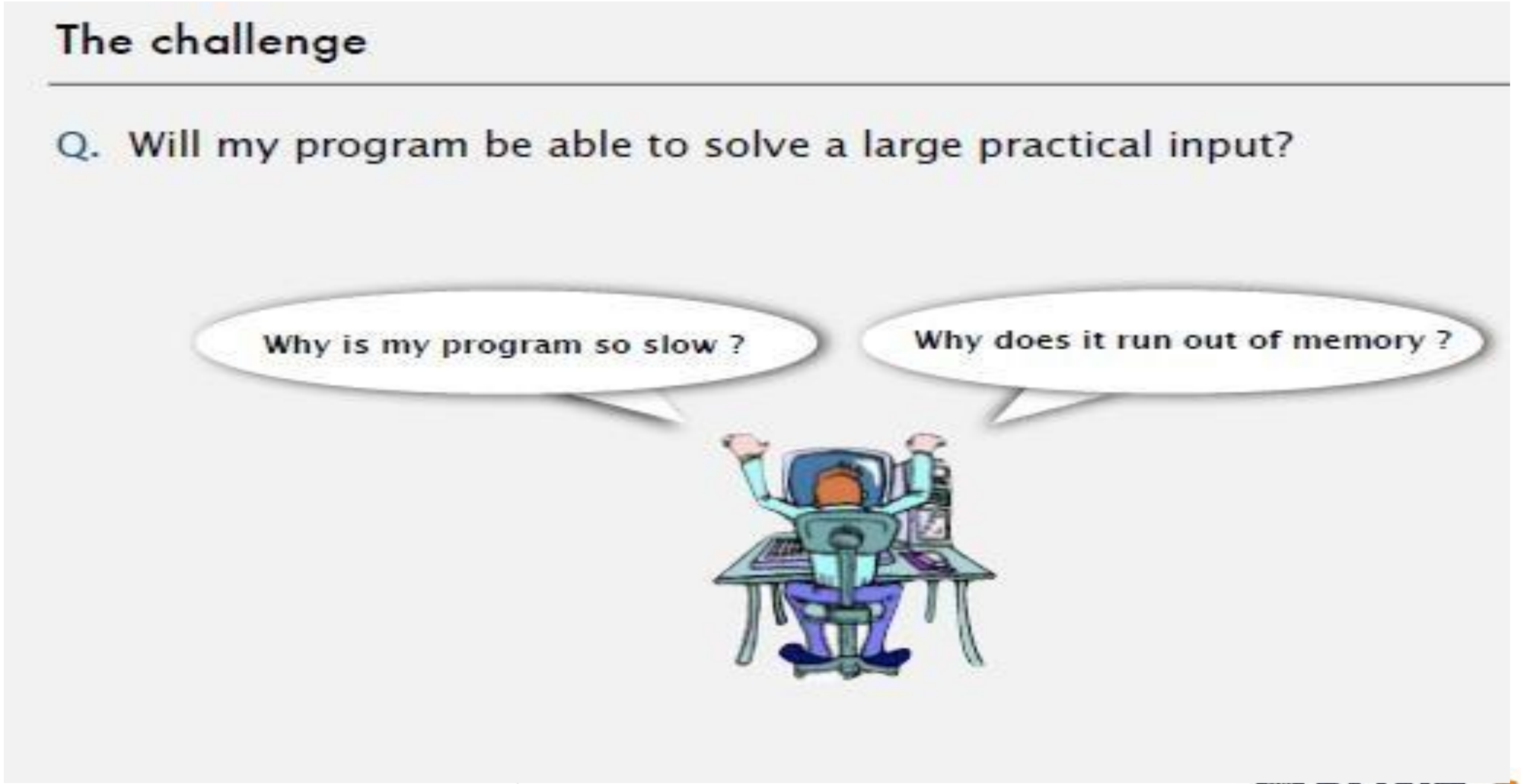
return Rsum(a,n-1)+a[n] .

} 2n+2 when n>0

RNSIT-CSE Bengaluru

Examples

```
{
    for i=1 to m do
        for j=1 to n
        do
            c[i,j]=a[i,j]+ b[i,j];
}
```



The challenge

Q. Will my program be able to solve a large practical input?

Why is my program so slow ?

Why does it run out of memory ?

Introduction- Order of Growth

- **Constant.** A program whose running time's order of growth is constant executes a fixed number of operations to finish its job; consequently its running time does not depend on N. – Accessing i^{th} element in an array
- **Logarithmic.** A program whose running time's order of growth is logarithmic is barely slower than a constant-time program
  - Binary search – Log N
- **Linear.** Programs that spend a constant amount of time processing each piece of input data, or that are based on a single for loop, are quite common.
  - The order of growth of such a program is said to be linear —its running time is proportional to N.
  - Searching for a key using sequential search

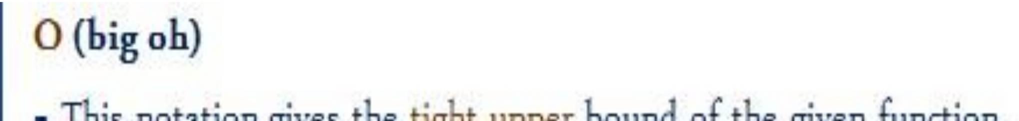
- **Linearithmic.** Algorithms whose running time for a problem of size N has order of growth N log N.
  - Merge sort, Quick Sort
- **Quadratic.** A typical program whose running time has order of growth N^2 has two nested for loops, used for some calculation involving all pairs of N elements.
  - Selection sort, Bubble sort, Insertion sort
- **Cubic.** A typical program whose running time has order of growth N^3 has three nested for loops, used for some calculation involving all triples of N elements.
  - Matrix multiplication, Floyd-Warshall algorithm
- **Exponential.** Programs whose running times are proportional to 2^N or N!
  - Exponential algorithms are extremely slow—you will never run one of them to completion for a large problem.

RNSIT-CSE
Bengaluru

- But they do exist at large !!!!
- it would take about **4 . 10^{10}** years for a computer making a trillion (**10^{12}**) operations per second to execute **2^{100}** operations
- **100!** operations, it is still longer than **4.5** billion *(***4.5 . 10^{9}***)* years— the estimated age of the planet Earth !!!!!

order of growth	name	typical code framework	description	example	T

Asymptotic Notations



O (big oh)

- This notation gives the tight upper bound of the given function

Problems

Prove the assertion $100n + 5 = O(n^2)$

$100n+5 = 100n +5n$

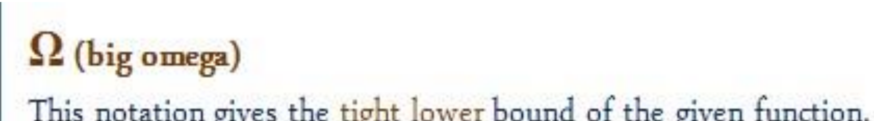
for all $n>=1 = 105n$

$c=105$

$n_0=1$

- $3n+2$

- $3n+3$
- $100n+6$
- $10n^2+4n+6$
- $1000n^2+100n-6$



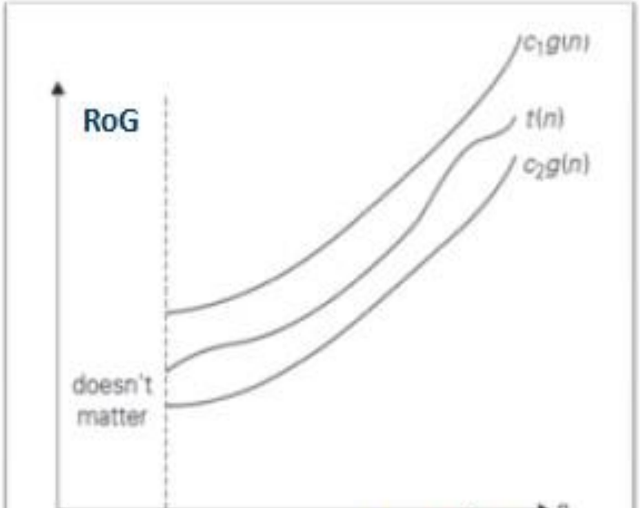
Ω (big omega)

This notation gives the tight lower bound of the given function.

- $n^3 \in \Omega(n^2)$, for $n \geq 0$ ($c=1$, $n_0=1$)
- $3n+2= \Omega (n)$
- $3n+3= \Omega (n)$
- $100n+6 =\Omega (n)$
- $10n^2 +4n +2= \Omega (n)$

Θ (big theta).

- This notation decides whether the upper and lower bounds of a given function (algorithm) are same or not..
- A function f(n) is said to be in Θ (g(n)), denoted f(n) $\in \Theta$ (g(n)), if f(n) is bounded both above and below by some positive constant multiples of g(n) for all large n,
  - i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that **$c_2g(n) \leq f(n) \leq c_1g(n)$ for all $n \geq n_0$.**



Examples

- $1/2\ n(n - 1) \in \Theta(n^2)$ (c1=1/2, c2=1/4, n0=2)
- $3n+2 \in \Theta(n)$ (c1=4, c2=3, n0=2)

Note: Analyze the algorithms at larger values of only,

- $3n+2 \in \Theta(n)$
- $3n+3 \in \Theta(n)$

Thank YOU

RNSIT-CSE
Bengaluru

MODULE- 1

(CONTI)

Presented By

Manjula L
Assistant Professor
Department of CSE
RNSIT

Asymptotic Notations(Continued)

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

(The analogous assertions are true for the Ω and Θ notations as well.)

PROOF (As you will see, the proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, and b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2 \max\{b_1, b_2\}$.) Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities. Adding the two inequalities above yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2\max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ■

So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a larger order of growth, i.e., its least efficient part:

$$\left.\begin{array}{|c|}\hline t_1(n) \in O(g_1(n)) \\ \hline t_2(n) \in O(g_2(n)) \\ \hline \end{array}\right\} \quad t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

USING LIMITS FOR COMPARING ORDER OF GROWTH

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n)\\ c>0 & \text{implies that } t(n) \text{ has the same order of growth as } g(n)\\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^{3}\end{cases}$$

IMPORTANT FORMULAS

- LH Hospital’s Rule:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

- Stirling's Formula:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \text{ for large values of } n.$$

Problems

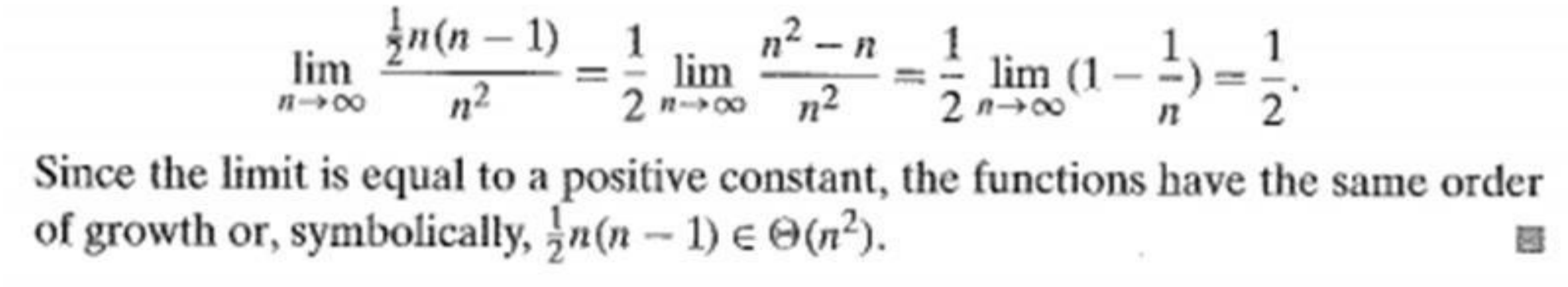
1. Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2.

2. Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$.

3. Compare the orders of growth of $n!$ and 2^n.

Solutions

1 .



$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^2-n}{n^2} = \frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right) = \frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1) \in \Theta(n^2)$. ■

Solutions

2.

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'}=\lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}=2\log_2 e\lim_{n\to\infty}\frac{\sqrt{n}}{n}=0.$$

Solutions

3.

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}=\lim_{n\to\infty}\sqrt{2\pi n}\frac{n^n}{2^n e^n}=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n=\infty.$$

Mathematical analysis -Non- recursive algorithms

Problem statement

Find the value of the largest element in a list of n numbers. (assume that the list is implemented as an array)

Input size ?

n

Basic operation ?

Number of comparison

ALGORITHM MaxElement (A[0..n − 1])

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
    maxval ←A[0]
    for i ←1 to n − 1 do   same of all arrays of if A[i]>maxval size n?
```

Mathematical analysis -Non- recursive algorithms

maxval←A[i] return maxval **YES !!!**

- No need to distinguish among the worst, average, and best cases.

General Plan

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input.
    - If it also depends on some additional property, the **worst-case**, **average-case**, and, if necessary, **best-case** efficiencies have to be investigated separately.

Mathematical analysis -Non- recursive algorithms

4. Set up a **sum** expressing the number of times the algorithm's **basic** operation is executed.
5. Using standard formulas and **rules** of **sum manipulation**, either find a closed form formula for the count or, at the very least, establish its order of growth.

Mathematical analysis -Non- recursive algorithms

RNSIT-CSE Bengaluru

Formulae that you may need !!!

1. $\log_a 1 = 0$
2. $\log_a a = 1$
3. $\log_a x^y = y \log_a x$
4. $\log_a xy = \log_a x + \log_a y$
5. $\log_a \frac{x}{y} = \log_a x - \log_a y$
6. $a^{\log_b x} = x^{\log_b a}$
7. $\log_a x = \frac{\log_b x}{\log_b a} = \log_a b \log_b x$

Sum Manipulation Formulae

1. $\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i$
2. $\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$
3. $\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i$, where $l \le m < u$
4. $\sum_{i=l}^{u} (a_i - a_{i-1}) = a_u - a_{l-1}$

Remember Combinatorics ?

Mathematical analysis -Non- recursive algorithms

1. Number of permutations of an n-element set: $P(n) = n!$
2. Number of k-combinations of an n-element set: $C(n, k) = \frac{n!}{k!(n-k)!}$
3. Number of subsets of an n-element set: 2^n

Mathematical analysis -Non- recursive algorithms

- **Summation Formulae**

Mathematical analysis -Non- recursive algorithms

1. $\sum_{i=l}^{u} 1 = \underbrace{1+1+\cdots+1}_{u-l+1 \text{ times}} = u - l + 1$ (l, u are integer limits, $l \le u$); $\sum_{i=1}^{n} 1 = n$

2. $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$

3. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$

4. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1}n^{k+1}$

5. $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1}-1}{a-1}$ $(a \ne 1)$; $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

6. $\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$

7. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + \gamma$, where $\gamma \approx 0.5772\ldots$ (Euler's constant)

8. $\sum_{i=1}^{n} \lg i \approx n \lg n$

Mathematical analysis -Non- recursive algorithms

```
ALGORITHM MaxElement (A[0..n − 1])
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A maxval
    ←A[0]
    for i ←1 to n − 1 do
        if A[i]>maxval
        maxval←A[i]
    return maxval
```

Let **C(n)** be number of times the basic operation is executed, hence

$C(n) = \sum_{i=1}^{n-1} 1 = n\text{-}1 - (1)+1 = n\text{-}1 = \boldsymbol{\Theta(n)}$

Mathematical analysis -Non- recursive algorithms

Element Uniqueness Problem

- Given an array, check whether all the elements in a given array of *n* elements are distinct

```
ALGORITHM UniqueElements(A[0..n − 1])
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns “true” if all the elements in A are distinct and “false” otherwise
    for i ←0 to n − 2 do    Input size ? for j ←i + 1 to n − 1 do
            if A[i]= A[j ] return falsen ←— — — — — —    return    trueBasic
    operation ?
```

Mathematical analysis -Non- recursive algorithms

Element Uniqueness Problem

- Given an array, check whether all the elements in a given array of *n* elements are distinct

```
ALGORITHM UniqueElements(A[0..n − 1])
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct and "false" otherwise
    for i ←0 to n − 2 do    Input size ? for j ←i + 1 to n − 1 do
            if A[i]= A[j ] return falsen ←                return    trueBasic
    operation ?
```

Mathematical analysis -Non- recursive algorithms

- Number of comparisons depends only on n? **No !!!**
    - It depends also on whether there are equal elements in the array ?

Mathematical analysis -Non- recursive algorithms

- What are possible worst-case inputs ?
  - i.e. the inner loop doesn't exit prematurely
- Array with no equal elements (distinct)
- Last two elements are only pair of equals

```
ALGORITHM UniqueElements(A[0..n − 1])
    for i ←0 to n − 2 do for j ←i + 1 to n − 1
    do if A[i]= A[j ] return false
    return true
```

$$\boxed{C_{worst}(n)=\sigma ni=-02\sigma nj=-i1+11} = \boxed{\sigma ni=-02[(n-1)-(i+1)+1]} = \boxed{\sigma ni=-02(n-1-i)}$$

$$= \boxed{\sigma_{i=0}^{n-2}(n-1) - \sigma_{i=0}^{n-2} i} = \boxed{(n-1)\,\sigma_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}}$$

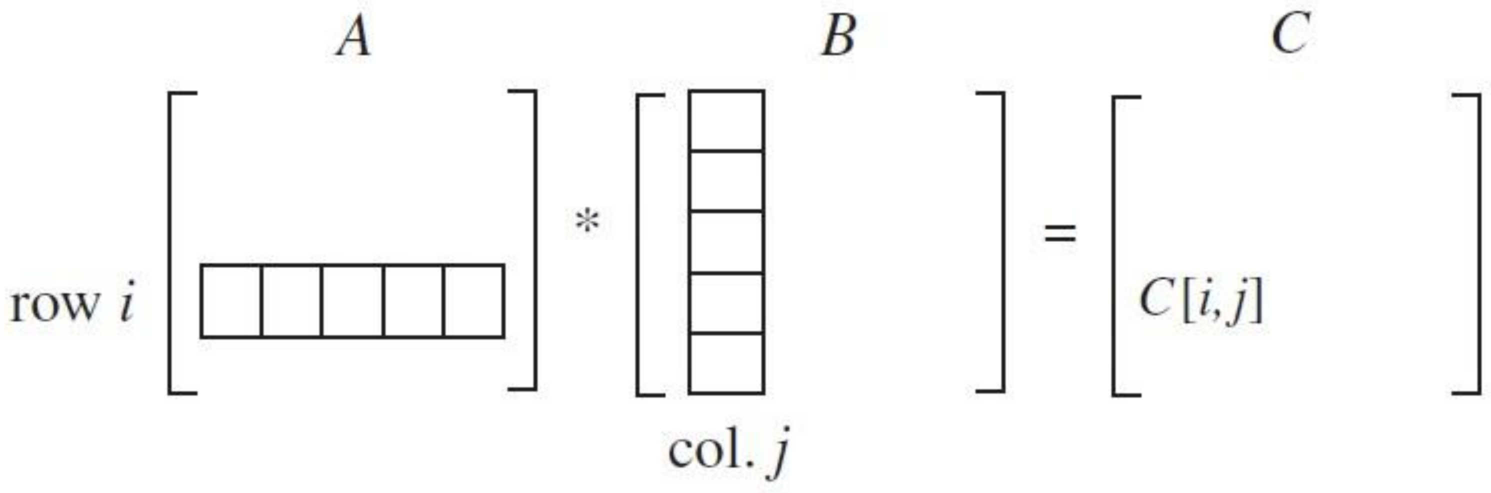
$$= \boxed{(n-1)^2 - \frac{(n-2)(n-1)}{2}} = \boxed{\frac{(n-1)n}{2}} \approx \boxed{\frac{1}{2}n^2 \in \theta(n^2)}$$

Matrix Multiplication

- Given two n × n matrices A and B, compute their product C=AB.

Mathematical analysis -Non- recursive algorithms

- C is an n × n matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:



- where $C[i, j] = A[i, 0]B[0, j] + \ldots + A[i, k]B[k, j] + \ldots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \le i, j \le n-1$.

Mathematical analysis -Non- recursive algorithms

```
ALGORITHM MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
    for i ←0 to n − 1 do
        for j ←0 to n − 1 do
            C[i, j ]←0.0
            for k←0 to n − 1 do
                C[i, j ]←C[i, j ]+ A[i, k] ∗ B[k, j]
    return C
```

Input size ?

Matrix order **n**

Basic operation ?

- Do we need to investigate worst case, best case and average case separately?

No !!!

- Let M(n) be total number of multiplications

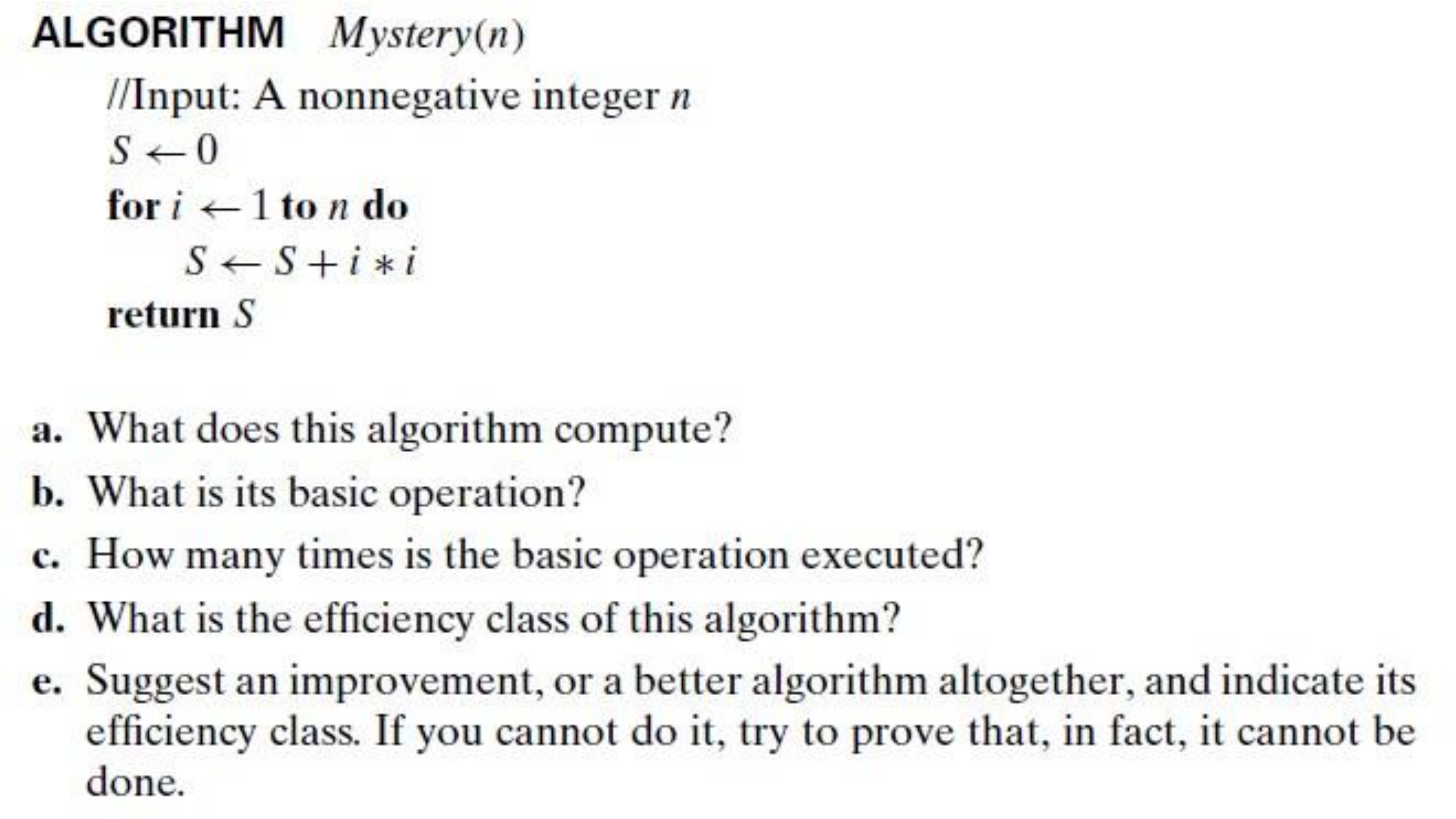
Mathematical analysis -Non- recursive algorithms

$M(n)$ σni=−01σnj=−01σnk=−011 = σni=−01σnj=−01n n−1n2 i=0 = n 3 == σ

Mathematical analysis -Non- recursive algorithms

Do It Yourself

Consider the following algorithm



```
ALGORITHM Mystery(n)
    //Input: A nonnegative integer n
    S ← 0
    for i ← 1 to n do
        S ← S + i * i
    return S
```

a. What does this algorithm compute?

b. What is its basic operation?

c. How many times is the basic operation executed?

d. What is the efficiency class of this algorithm?

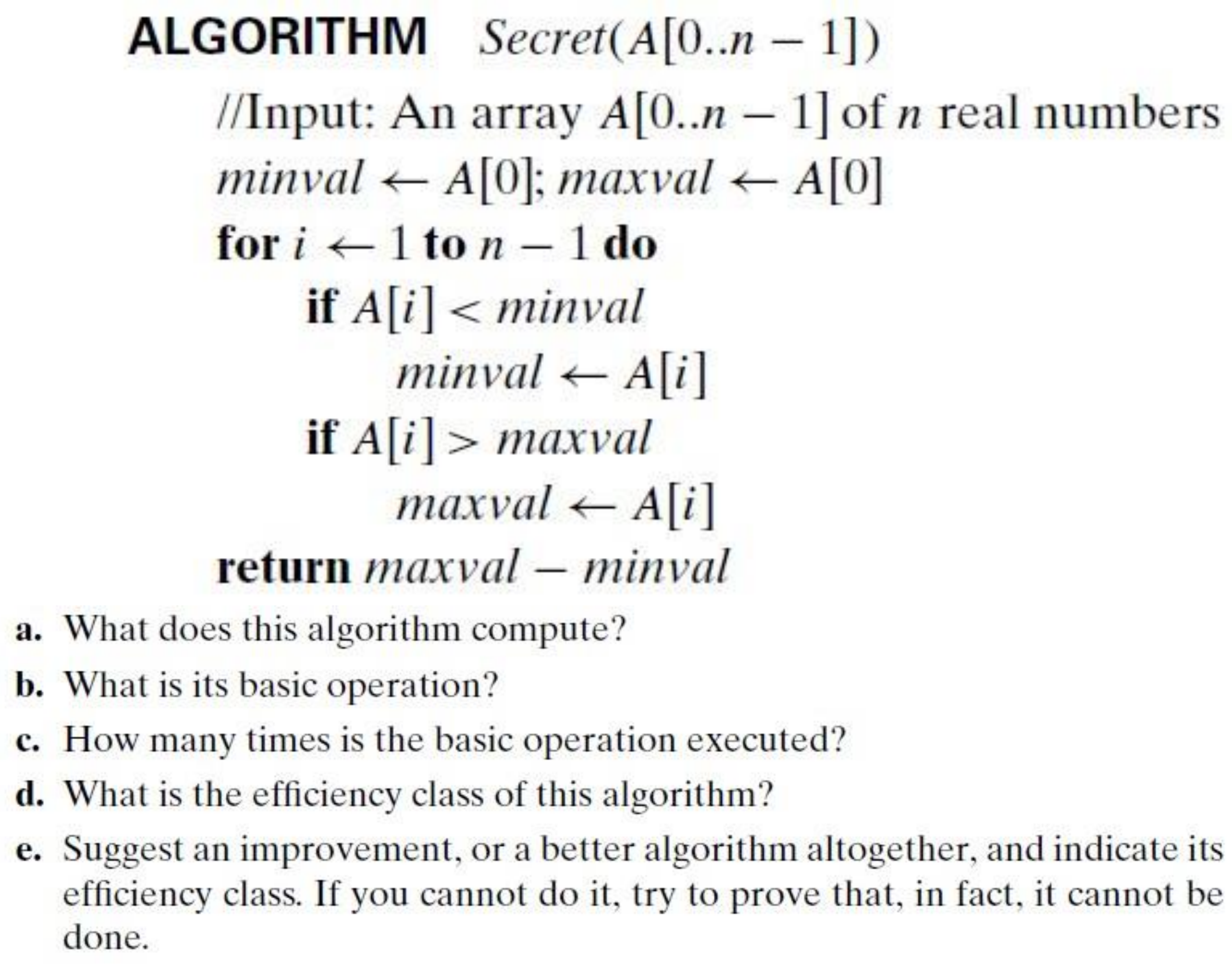
e. Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

Mathematical analysis -Non- recursive algorithms

Do It Yourself

Consider the following algorithm

Mathematical analysis -Non- recursive algorithms



```
ALGORITHM  Secret(A[0..n − 1])
    //Input: An array A[0..n − 1] of n real numbers
    minval ← A[0]; maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] < minval
            minval ← A[i]
        if A[i] > maxval
            maxval ← A[i]
    return maxval − minval
```

a. What does this algorithm compute?

b. What is its basic operation?

c. How many times is the basic operation executed?

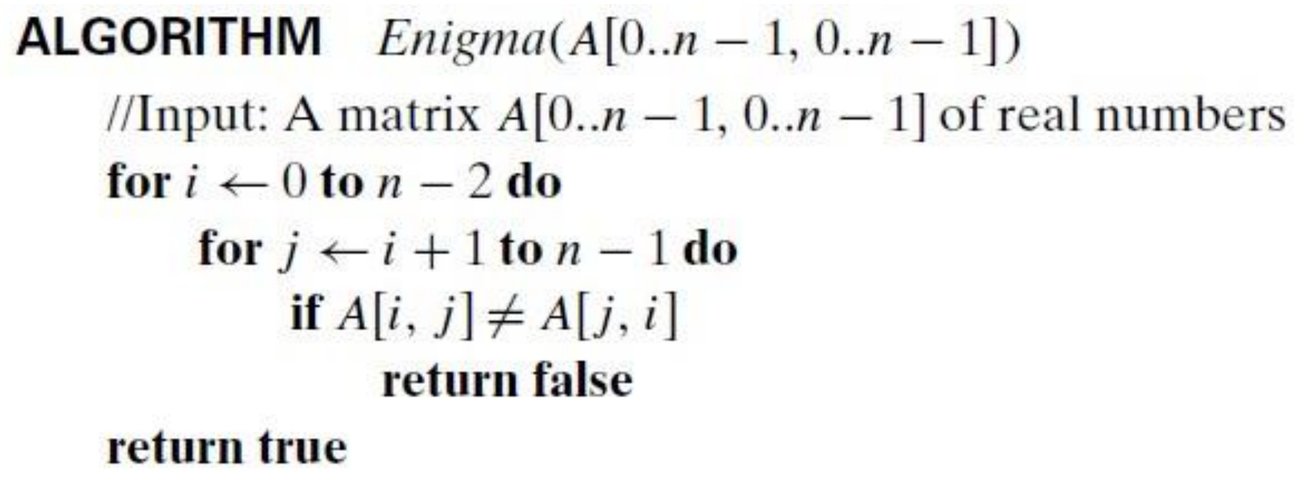
d. What is the efficiency class of this algorithm?

e. Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

Mathematical analysis -Non- recursive algorithms

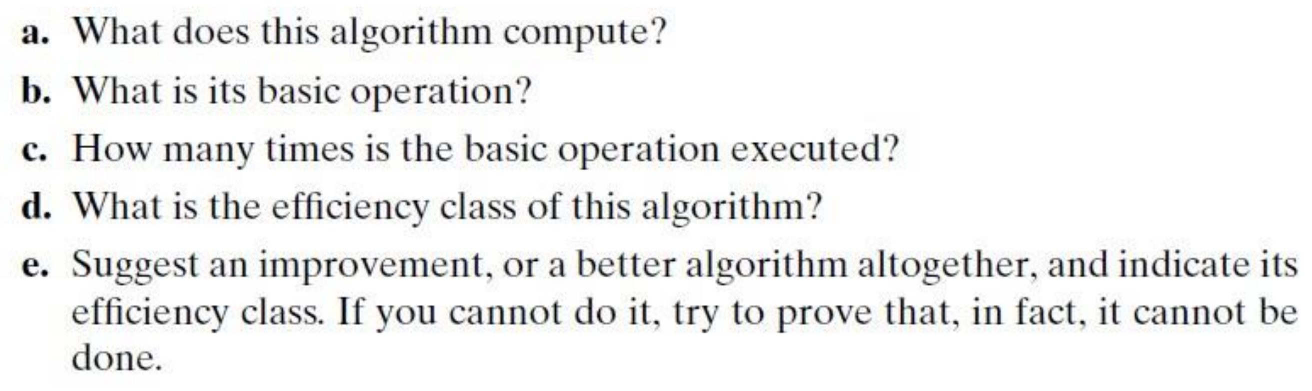
Do It Yourself

Consider the following algorithm



```
ALGORITHM  $Enigma(A[0..n-1, 0..n-1])$
    //Input: A matrix $A[0..n-1, 0..n-1]$ of real numbers
    for $i \leftarrow 0$ to $n-2$ do
        for $j \leftarrow i+1$ to $n-1$ do
            if $A[i, j] \neq A[j, i]$
                return false
    return true
```

Mathematical analysis -Non- recursive algorithms



a. What does this algorithm compute?

b. What is its basic operation?

c. How many times is the basic operation executed?

d. What is the efficiency class of this algorithm?

e. Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

Mathematical analysis - Recursive algorithms

- Compute the factorial function F(n) = n! for an arbitrary nonnegative integer n.
- i.e. n!= 1 . . . . . (n − 1) . n = (n − 1)! . n for n ≥ 1 and 0!= 1 by definition
- i.e. F(n) = F(n − 1) . n
- Following is the recursive algorithm that computes the factorial (n)

ALGORITHM F(n)

//Computes n! recursively

//Input: A nonnegative integer n

//Output: The value of n!

Input size ?

n

Basic operation ?

Mathematical analysis - Recursive algorithms

if n = 0 return 1 **Multiplication** else return F(n − 1) n

*

- Let M(n) be the number of executions of the basic operation.
- F(n) is computed according to the formula

F(n) = F(n − 1) . n for n > 0, ▪ The number of multiplications M(n) is given by

```
ALGORITHM F(n)
    if n = 0 return 1 else
    return F(n − 1) ∗ n
```

$$M(n)=M(n-1) + 1 \quad \text{for } n>0$$

to compute F(n−1)   to multiply F(n−1) by n

- M(n − 1) multiplications are spent to compute F(n − 1), and one more multiplication is needed to multiply the result by n.
- The equation defines the sequence M(n) that we need to find.

Mathematical analysis - Recursive algorithms

- This equation defines M(n) not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely $n - 1$.
- Such equations are called recurrence relations or, for brevity, recurrences
- To solve a recurrence relation say $M(n)=M(n-1) + 1$ for $n>0$, you need an initial condition that tells us the value with which the sequence starts.
- Observe that this value can be obtained by inspecting the condition that makes the algorithm stop its recursive calls: If $n=0$ return 1 . This tells us two things
    - First, since the calls stop when $n = 0$, the smallest value of n for which this algorithm is executed and hence M(n) defined is 0

```
ALGORITHM F(n)
    if n = 0 return 1 else
    return F(n − 1) ∗ n
```

Mathematical analysis - Recursive algorithms

- Second, by inspecting the pseudocode's exiting line, we can see that when $n = 0$, the algorithm performs no multiplications $M(0)=0$

The call stops when $n=0$ no multiplications when $n = 0$

- The resulting recurrence relation with initial condition is given by

$$M(n) = M(n - 1) + 1 \text{ for } n > 0, M(0) = 0.$$

- Observe two functions
  - The first is the factorial function $F(n)$ itself; it is defined by the recurrence $F(n) = F(n - 1) . n$ for every $n > 0$, $F(0) = 1$.
  - The second is the number of multiplications $M(n)$ needed to compute $F(n)$ by the recursive algorithm

Mathematical analysis - Recursive algorithms

$M(n) = M(n-1) + 1$ for $n > 0$, $M(0) = 0$.

This is to be solved

Solve the following recurrence relation (Backward Substitution)

$M(n) = M(n-1) + 1$ for $n > 0$, $M(0) = 0$.

Solution:

$M(n) = M(n-1) + 1$ Substitute n-1 in place of n

$= [M(n-2) + 1] + 1 = M(n-2) + 2$ Substitute n-1 in place of n $=$ $[M(n-3) + 1] + 2 = M(n-3) + 3$ and so on…

Mathematical analysis - Recursive algorithms

Can you onbserve an emerging pattern ? **M(n-i)+i**

$M(n) = M(n - 1) + 1 = \ldots = M(n - i) + i = \ldots = M(n - n) + n = n$. As $(M(0)=0)$

General plan

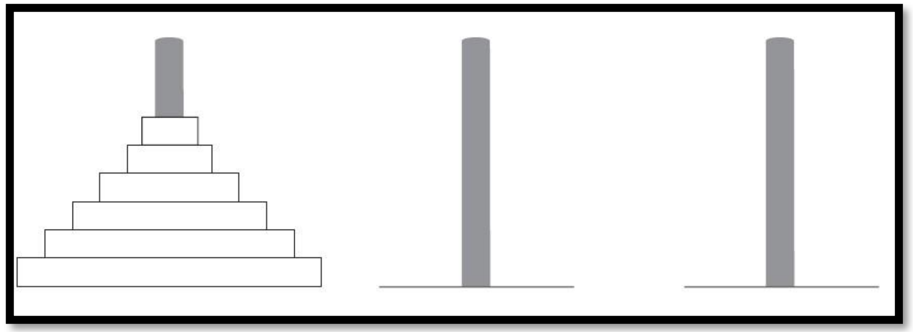
1. Decide on a parameter (or parameters) indicating an input’s size.
2. Identify the algorithm’s basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size;
    1. if it can, the **worst-case**, **average-case**, and **best-case** efficiencies must be investigated separately.

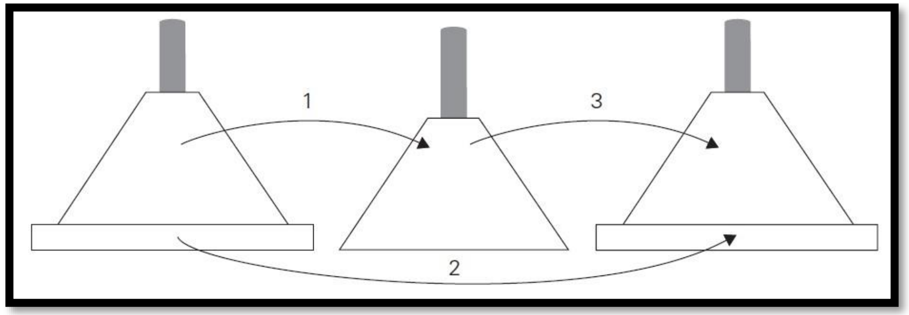
Mathematical analysis - Recursive algorithms

4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

Tower of Hanoi Puzzle

- Given n disks of different sizes, goal is to move/slide all of them onto any of three pegs with the help of second peg as an auxiliary .
- The constraints are
  - Move **only one** disk at a time
  - Larger disk can not be placed on smaller one
- Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top.





TowerofHanoi Puzzle

- This problem has a elegant recursive solution
- To move n>1 disks from peg 1 to peg 3 (with peg 2 as auxiliary),
  - we first move recursively n – 1 disks from peg 1 to peg 2 (with peg 3 as auxiliary),
- Then move the largest disk directly from peg 1 to peg 3, and,
  - finally, move recursively n – 1 disks from peg 2 to peg 3 (using peg 1 as auxiliary).
  - Of course, if n = 1, we simply move the single disk directly from the source peg to the destination peg

Mathematical analysis - Recursive algorithms

Recurrence relation and the solution approach

- Input size ?
  - Number of discs i.e., **n**
- Basic operation ?
  - Moving one disc
- Hence the number of moves M(n) depends only on **n**
- The recurrence relation is given by

$$M(n) = M(n - 1) + 1 + M(n - 1) \text{ for } n > 1.$$

- What is the initial condition ?
  - $M(1)=1$
- Hence the obvious recurrence relation is givne by

$$M(n) = 2M(n - 1) + 1 \text{ for } n > 1,$$
$$M(1) = 1.$$

Mathematical analysis - Recursive algorithms

Solve the recurrence relation

$$M(n) = 2M(n-1) + 1 \text{ for } n > 1,$$

$M(1) = 1$. **Solution**

$M(n) = 2M(n-1) + 1$ substitute $M(n-1) = 2M(n-2) + 1$

$= 2[2M(n-2) + 1] + 1 = 2^2M(n-2) + 2 + 1$ substitute $M(n-2) = 2M(n-3) + 1 =$

$2^2[2M(n-3) + 1] + 2 + 1 = 2^3M(n-3) + 2^2 + 2 + 1.$

Can you guess the next one ?

$$2^4M(n-4) + 2^3 + 2^2 + 2 + 1,$$

After i substitutions we get

$M(n) = 2^iM(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 2 + 1 = 2^iM(n-i) + 2^i - 1.$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$,

$$M(n) = 2^{n-1}M(n-(n-1)) + 2^{n-1} - 1$$

$$= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = \mathbf{2^{n} - 1.}$$

Important problem types

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

Sorting

- The **sorting problem** is to rearrange the items of a given list in nondecreasing order
- As a practical matter, we usually need to sort
    - lists of numbers,

Important problem types

- characters from an alphabet,
- character strings, and, most important,
  - records similar to those maintained by schools about their students,
  - libraries about their holdings, and
  - companies about their employees

- In the case of records, we need to choose a piece of information to guide sorting
  - Such a specially chosen piece of information is called a **key**.

Sorting

- Sorting helps searching !!!!
- Although some algorithms are indeed better than others, there is no algorithm that would be the best solution in all situations.
  - Some of the algorithms are simple but relatively slow,
  - while others are faster but more complex;

Important problem types

- Some work better on randomly ordered inputs,
- while others do better on almost-sorted lists;
- some are suitable only for lists residing in the fast memory,
- while others can be adapted for sorting large files stored on a disk; and so on.

Sorting

Properties of sorting algorithms

- A sorting algorithm is called **stable** if it preserves the relative order of any two equal elements in its input
- In other words, if an input list contains two equal elements in positions i and j where i < j, then in the sorted list they have to be in positions i* and j* respectively, such that i* < j*

Important problem types

- This property can be desirable if, for example, we have a list of students sorted alphabetically and we want to sort it according to student GPA:
  - a stable algorithm will yield a list in which students with the same GPA will still be sorted alphabetically.
- Generally speaking, algorithms that can exchange keys located far apart are not stable, but they usually work faster

Sorting

Properties of sorting algorithms

- A sorting algorithm is called **in-place** if it does not require extra memory, except, possibly, for a few memory units

Important problem types

Sorting Better Example for stable sort

Considerthe following example of student names and their respective class

$(Dave, A)$
$(Alice, B)$
(Ken, A)
$(Eric, B)$
$(Carol, A)$

Sort the data according to names, the sorted list will not be grouped according sectionto sections

$(Alice, B)$
$(Carol, A)$
$(Dave, A)$
$(Eric, B)$
(Ken, A)

Sort again to obtain list of students section wise too. The dataset is now sorted !!!!according to sections, but not according to names.

$(Carol, A)$
$(Dave, A)$
(Ken, A)
$(Eric, B)$
$(Alice, B)$

In the name sorted list the tuple (Alice, B) was before (Eric, B)

Important problem types

A stable sorting algorithm

would result in

$(Carol, A)$
$(Dave, A)$
(Ken, A)
$(Alice, B)$
$(Eric, B)$

- **Searching**
- The ***searching problem*** deals with finding a given value, called a ***search key***, in a given set (or a multiset, which permits several elements to have the same value).
  - Sequential / linear
  - Binary search (sorted input is a prerequisite but very efficient for large database)

Important problem types

String processing

- A ***string*** is a sequence of characters from an alphabet.
- Strings of particular interest are text strings which comprise letters, numbers, and special characters;
  - bit strings, which comprise zeros and ones; and
  - gene sequences, which can be modeled by strings of characters from the fourcharacter alphabet **{A, C, G, T}**
- One particular problem—that of searching for a given word in a text—has attracted special attention from researchers
- **Graph problems**
- a ***graph*** can be thought of as a collection of points called **vertices**, some of which are connected by line segments called **edges**.

Important problem types

- Graphs can be used for modeling a wide variety of applications, including
  - transportation,
  - communication, social and economic networks,
  - project scheduling, and
  - games

Combinatorial Problems

- These are problems that ask, explicitly or implicitly, to find a combinatorial object—such as a permutation, a combination, or a subset—that satisfies certain constraints.

Numerical Problems

- ***Numerical problems***, another large special area of applications, are problems that involve mathematical objects of continuous nature:

Important problem types

- solving equations and systems of equations,
- computing definite integrals,
- evaluating functions, and so on.

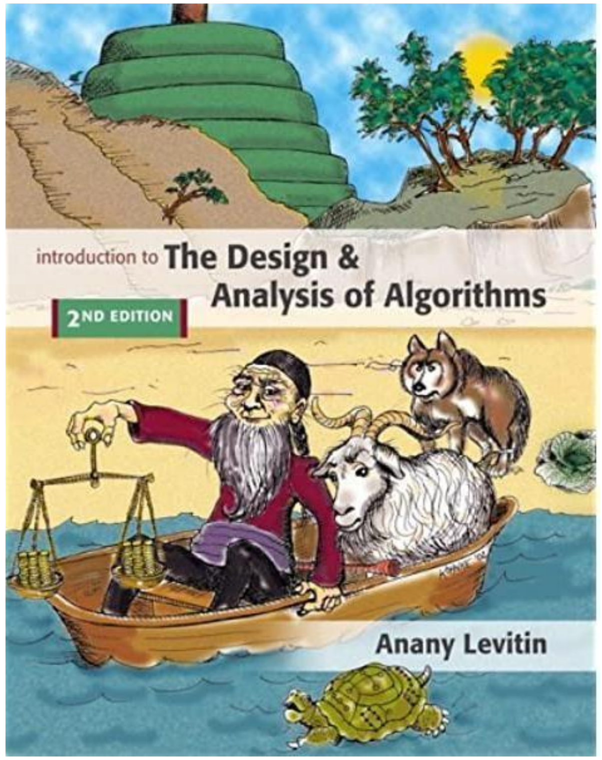
ESTD:2001
An Institute with a Difference

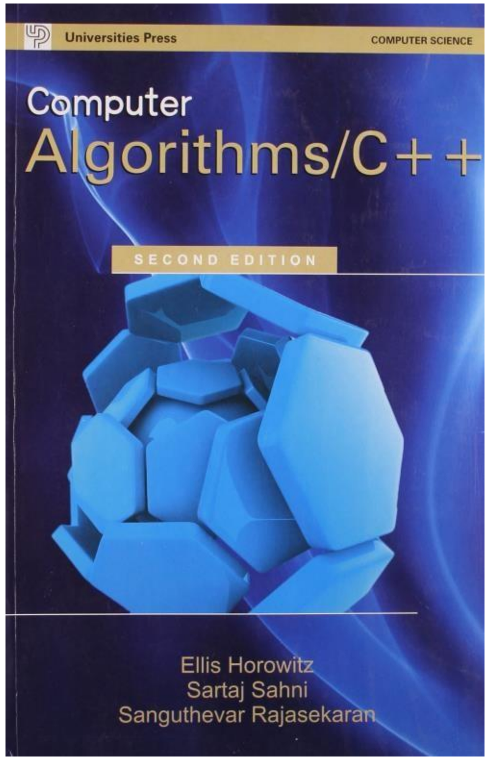
Design and Analysis of Algorithms

Divide and Conquer

Manjula L
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

Text Books





- Divide-and-Conquer (DaC) is probably the best-known general algorithm design technique.

Introduction

- Given a function to compute on n input the DaC approach suggests splitting the inputs into k distinct subsets,1< k < n, yielding k subproblems
- These subproblem must be solved and then a method must be found to combine solutions into a solution of the whole.
- If the sub problems are relatively large then the divide and conquer approach can possibly be reapplied
- Often the sub problems resulting from our divide and conquer design are of the same type as the original problem.
- Fur those cases the re application of the divide and conquer principle is naturally expressed by a recursive algorithm

- The smaller and smaller sub problems of the same kind are generated until eventually sub problems that are small enough to be solved without splitting are produced

Introduction

Example : Detecting a counterfeit coin

- Given a bag of n coins and a machine that weighs 2 sets of coins, the task is to find
  - Whether the bag contains a counterfeit coin
  - If present then identify the Counterfeit coin

Introduction

Control abstraction for Divide and Conquer approach

```
Algorithm DAndC(P)
    {
        if Small(P) then return S(P)
        else
        {
            divide P into Smaller instances P1, P2, P3  ... Pk,  k≥1 ;
            Apply DAndC to each of these subproblems;
            return Combine(DAndC(P1), DAndC(P2),...DAndC(Pk));
        }
    }
```

Boolean valued function that determines whether the input is small enough or not

Solution to the problem

A function that determines the solution to **P** using the solutions to **k** subproblems

Introduction

- If the size of **P** is **n**, And the sizes of the **k** subproblems are **n1, n2, n3… nk**, respectively, then the computing time of DAndC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \dots + T(nk) & \text{otherwise} \end{cases}$$

Where,

- **T(n)** is the time for DAndC on any input of size n
- **g(n)** is the time to compute the answer directly for small inputs

Introduction

- **f(n)** is the time for dividing **P** and combining the solutions to subproblems.

▪ The complexity of many divide and conquer is given by recurrences of the form

▪ $$T(n) = \begin{cases} T(1) & n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases}$$

Where,

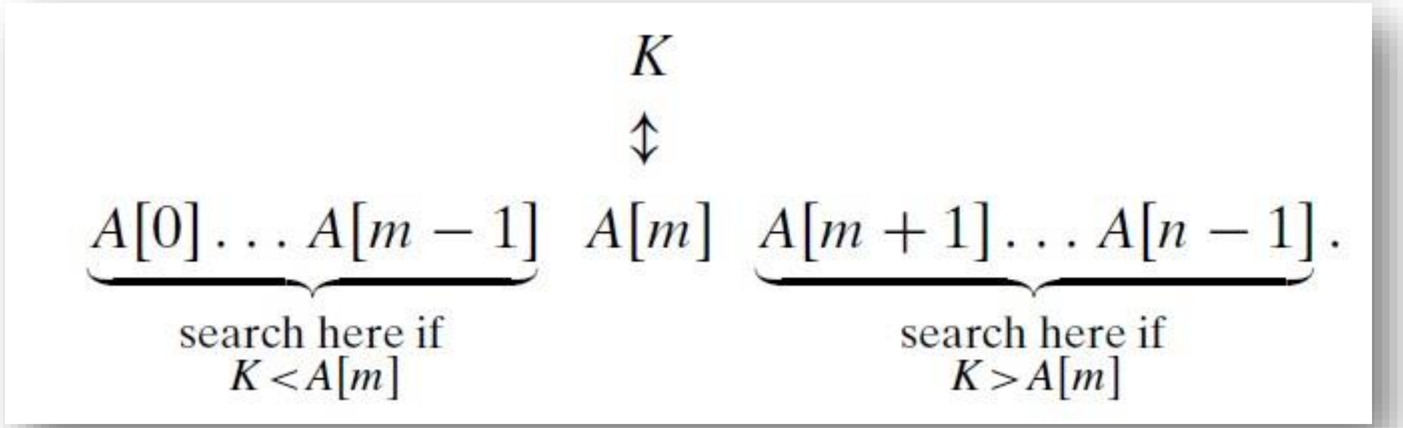
- **a** and **b** are constants

Introduction

- **T(1)** is known
- **n** is a power of **b** (i.e., **$n=b^k$**)

Binary Search

- Binary search is a remarkably efficient algorithm for searching in a sorted array
- It works by comparing a search key **K** with the array's middle element **A[m]**.
- If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if **K <A[m]**, and for the second half if

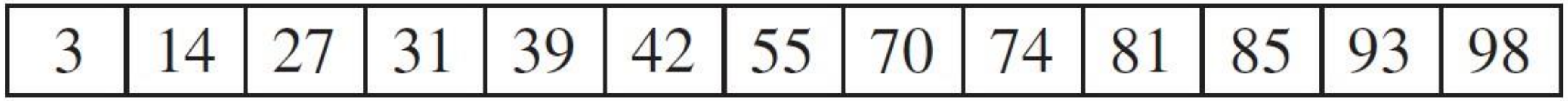

$$\begin{array}{c} K \\ \updownarrow \\ \underbrace{A[0]\ldots A[m-1]}_{\substack{\text{search here if}\\ K<A[m]}}\ A[m]\ \underbrace{A[m+1]\ldots A[n-1]}_{\substack{\text{search here if}\\ K>A[m]}}. \end{array}$$

K >A[m]

Binary Search

Example

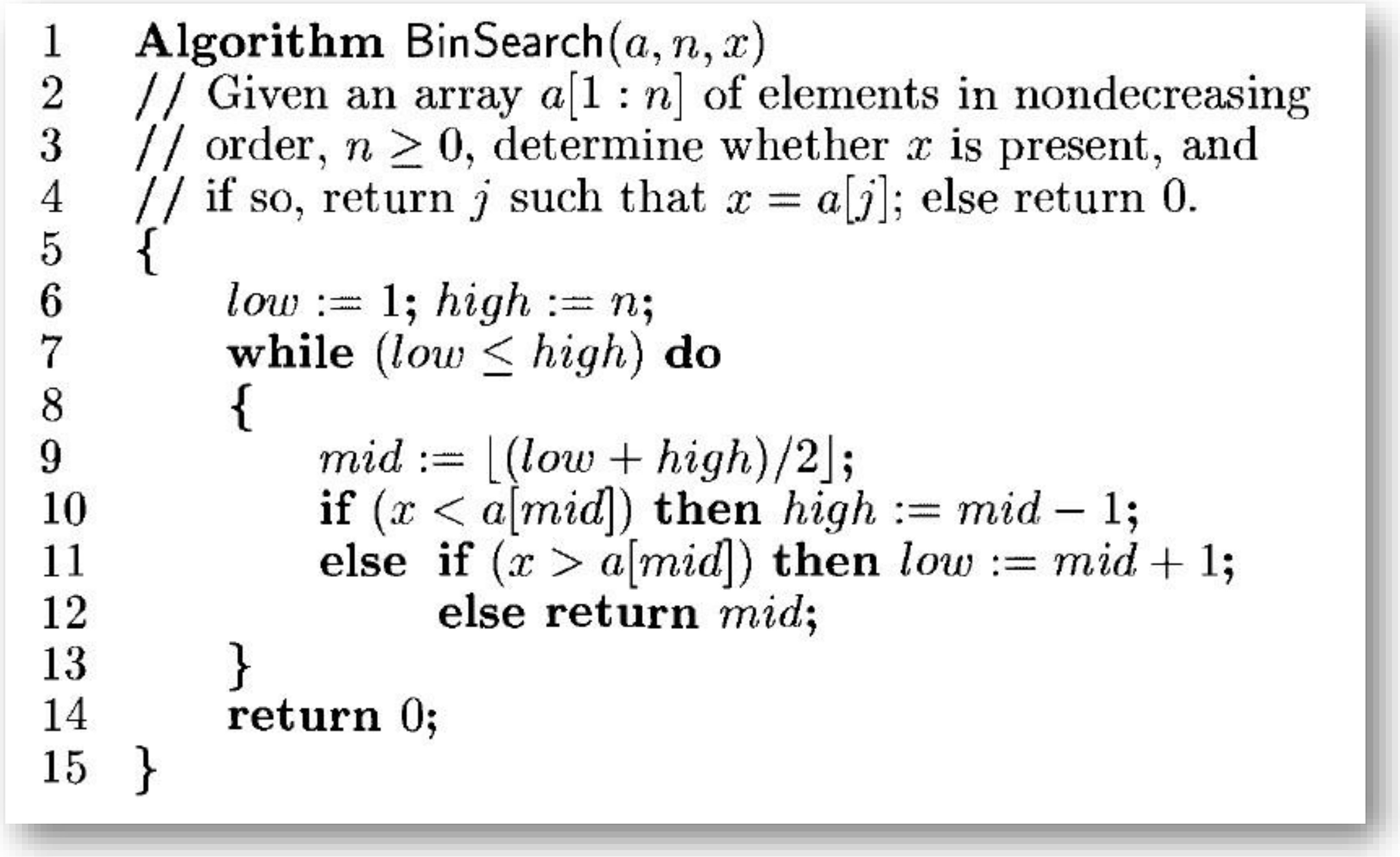
- Apply binary search algorithm to the following set of numbers considering **70** as the key



3	14	27	31	39	42	55	70	74	81	85	93	98

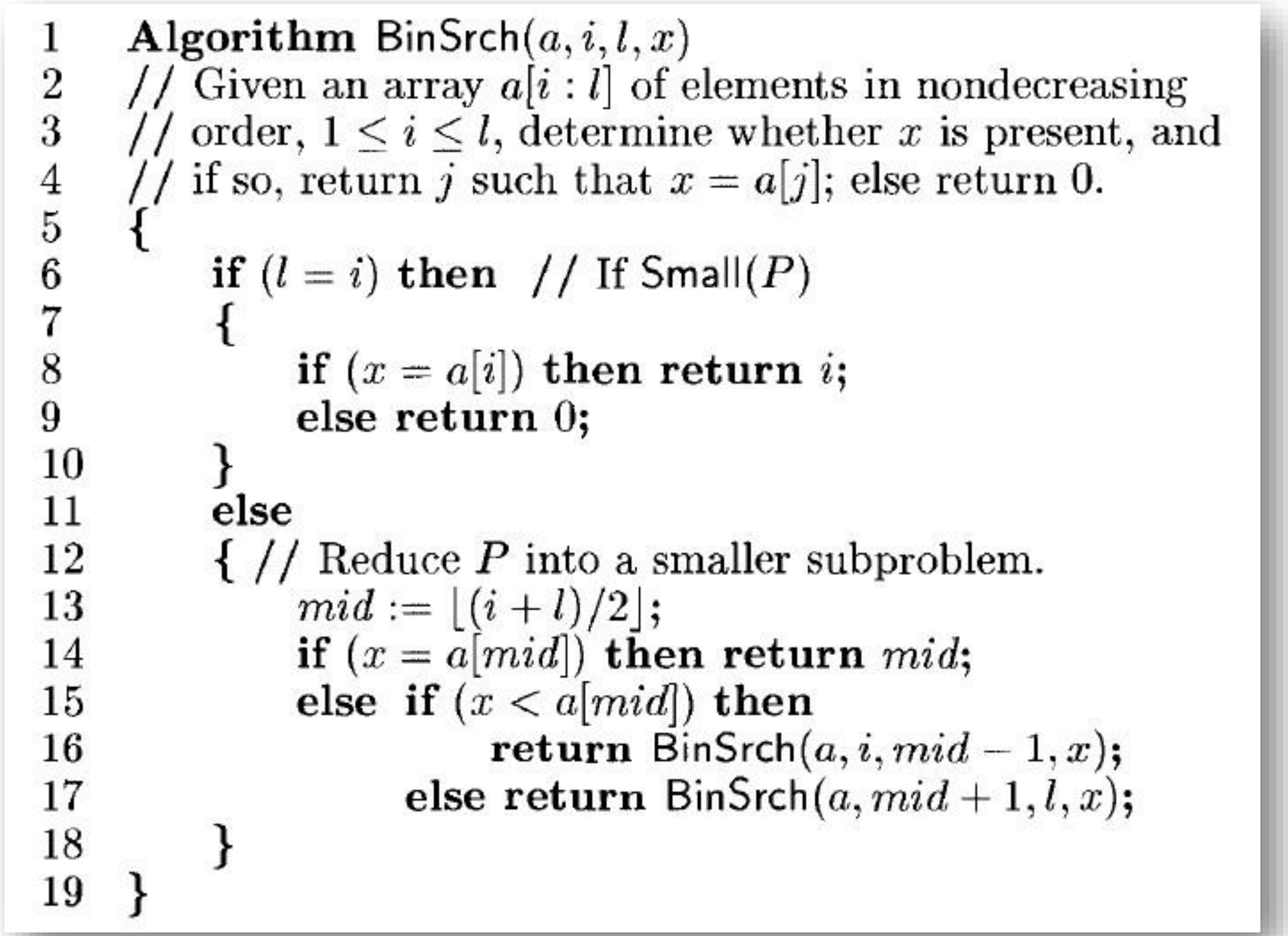
index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	*l*						*m*						*r*
iteration 2								*l*		*m*			*r*
iteration 3								*l*,*m*	*r*				

Binary Search : Non-Recursive



```
Algorithm BinSearch($a, n, x$)
// Given an array $a[1:n]$ of elements in nondecreasing
// order, $n \geq 0$, determine whether $x$ is present, and
// if so, return $j$ such that $x = a[j]$; else return 0.
{
    $low := 1$; $high := n$;
    while ($low \leq high$) do
    {
        $mid := \lfloor (low + high)/2 \rfloor$;
        if ($x < a[mid]$) then $high := mid - 1$;
        else if ($x > a[mid]$) then $low := mid + 1$;
             else return $mid$;
    }
    return 0;
}
```

Binary Search : Recursive



```
Algorithm BinSrch($a, i, l, x$)
// Given an array $a[i : l]$ of elements in nondecreasing
// order, $1 \le i \le l$, determine whether $x$ is present, and
// if so, return $j$ such that $x = a[j]$; else return 0.
{
    if ($l = i$) then  // If Small($P$)
    {
        if ($x = a[i]$) then return $i$;
        else return 0;
    }
    else
    { // Reduce $P$ into a smaller subproblem.
        $mid := \lfloor (i + l)/2 \rfloor$;
        if ($x = a[mid]$) then return $mid$;
        else if ($x < a[mid]$) then
                  return BinSrch($a, i, mid - 1, x$);
             else return BinSrch($a, mid + 1, l, x$);
    }
}
```

Binary Search : Analysis

Recurrence Relation

$$T(n) = \begin{cases} 1 & n=1 \\ T\left(\frac{n}{2}\right)+1 & n>1 \end{cases}$$

Solution

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

$= [T(n/4)+1]+1 = T(n/4)+2$

$=[T(n/8)+1]+2 = T(n/8) +3$ -

- - - - - - - - - - - - - -

- $=T(n/2^k)+k$ $n =b^k$, $n=2^k$, $\log n= \log 2^k$, $k=\log n$
- $=T(n/n)+k$
- $=1+k$

- $=1+\log n$
- **$T(n)=O(\log n)$**

Merge Sort

- Merge sort is a perfect example of a successful application of the divide-and conquer technique.
- It has the nice property that in the worst case its complexity is **O(nlogn).**
- Let us assume that the set of elements are to be sorted in **non-decreasing** order that is in **ascending** order.
- Given a sequence of n elements , a[1]……a[n], the idea is to imagine them split into two sets a[1]…… a[$\lfloor n/2 \rfloor$] and a[$\lfloor n/2 \rfloor$+1]…… a[n].
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of **n** elements
- This is an ideal example of the divide-and-conquer strategy in which the splitting is into two equal-sized sets and the combining operation is the merging of two sorted sets into one

Merge Sort

Divide

Divide

Divide

Divide

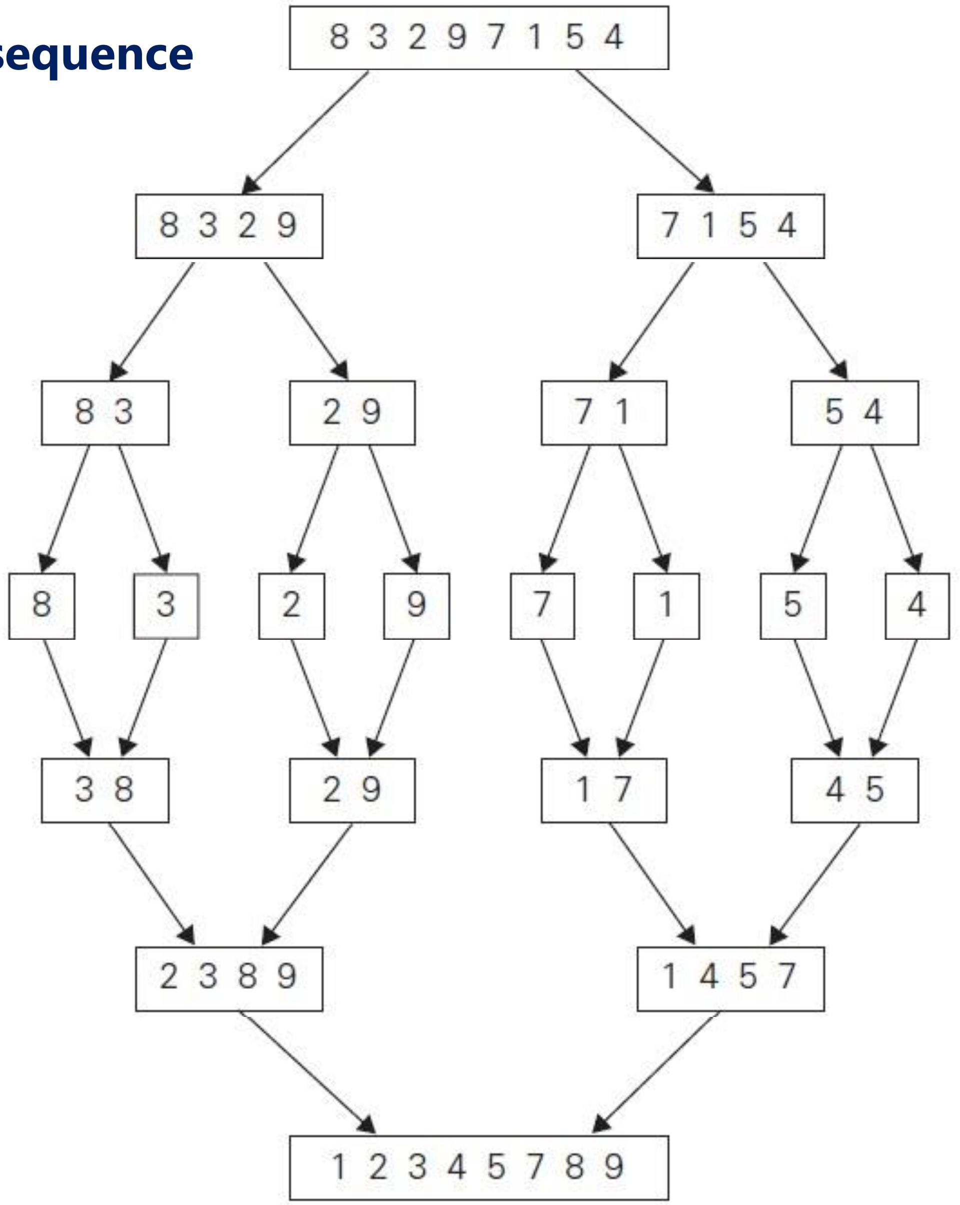
Combine

Combine

Merge Sort

Input sequence

sequence



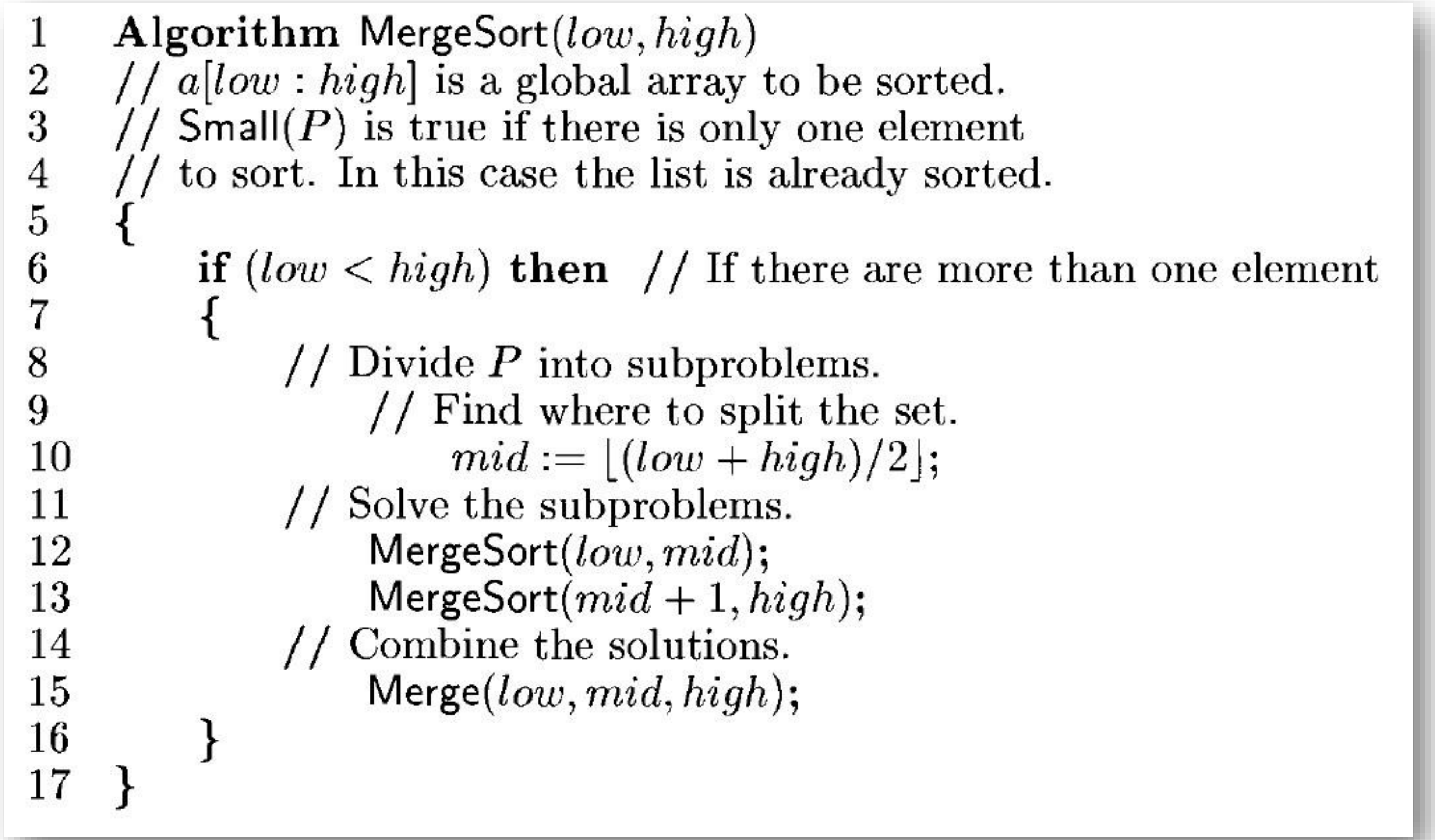
Merge Sort

Do it yourself

- Consider the input sequence

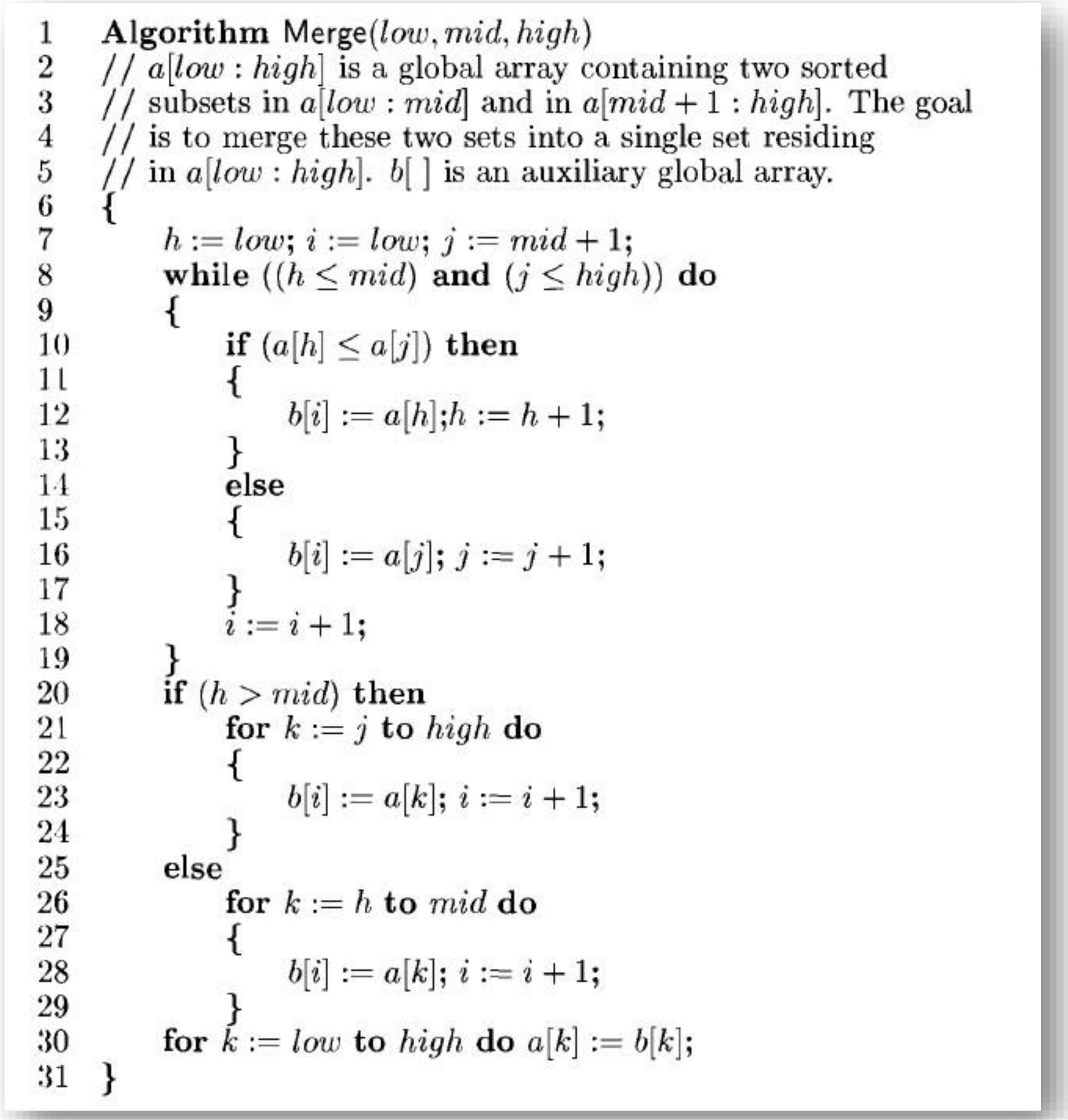
11, 44, 22, 99, 66, 33, 88, 55, 77, 00

Obtain the merge sort tree representation showing the divide and combine phase



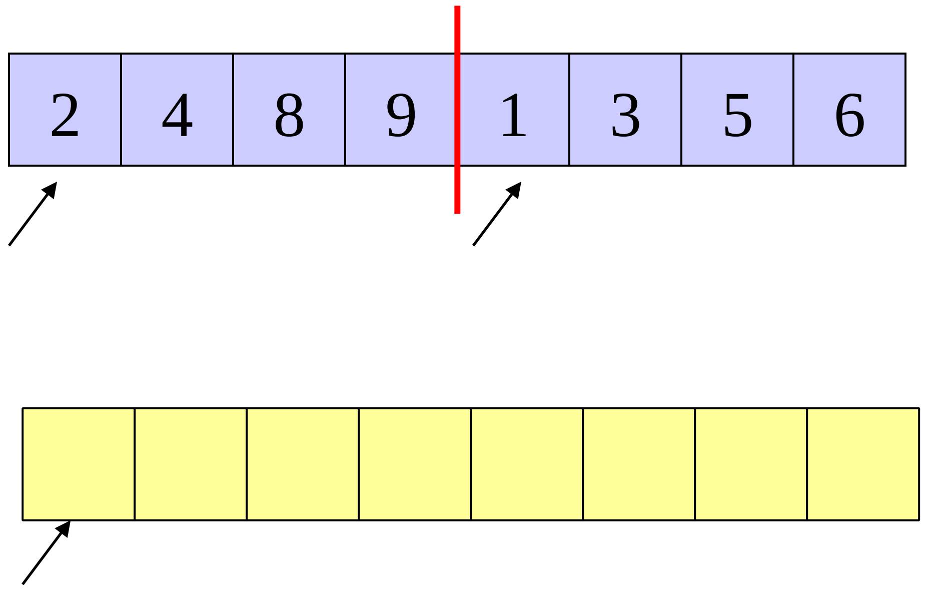
```
Algorithm MergeSort($low, high$)
// $a[low : high]$ is a global array to be sorted.
// Small($P$) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if ($low < high$) then  // If there are more than one element
    {
        // Divide $P$ into subproblems.
            // Find where to split the set.
                $mid := \lfloor (low + high)/2 \rfloor$;
        // Solve the subproblems.
            MergeSort($low, mid$);
            MergeSort($mid + 1, high$);
        // Combine the solutions.
            Merge($low, mid, high$);
    }
}
```

Working of Merge



```
Algorithm Merge($low, mid, high$)
// $a[low : high]$ is a global array containing two sorted
// subsets in $a[low : mid]$ and in $a[mid + 1 : high]$. The goal
// is to merge these two sets into a single set residing
// in $a[low : high]$. $b[\ ]$ is an auxiliary global array.
{
    $h := low$; $i := low$; $j := mid + 1$;
    while (($h \leq mid$) and ($j \leq high$)) do
    {
        if ($a[h] \leq a[j]$) then
        {
            $b[i] := a[h]$;$h := h + 1$;
        }
        else
        {
            $b[i] := a[j]$; $j := j + 1$;
        }
        $i := i + 1$;
    }
    if ($h > mid$) then
        for $k := j$ to $high$ do
        {
            $b[i] := a[k]$; $i := i + 1$;
        }
    else
        for $k := h$ to $mid$ do
        {
            $b[i] := a[k]$; $i := i + 1$;
        }
    for $k := low$ to $high$ do $a[k] := b[k]$;
}
```

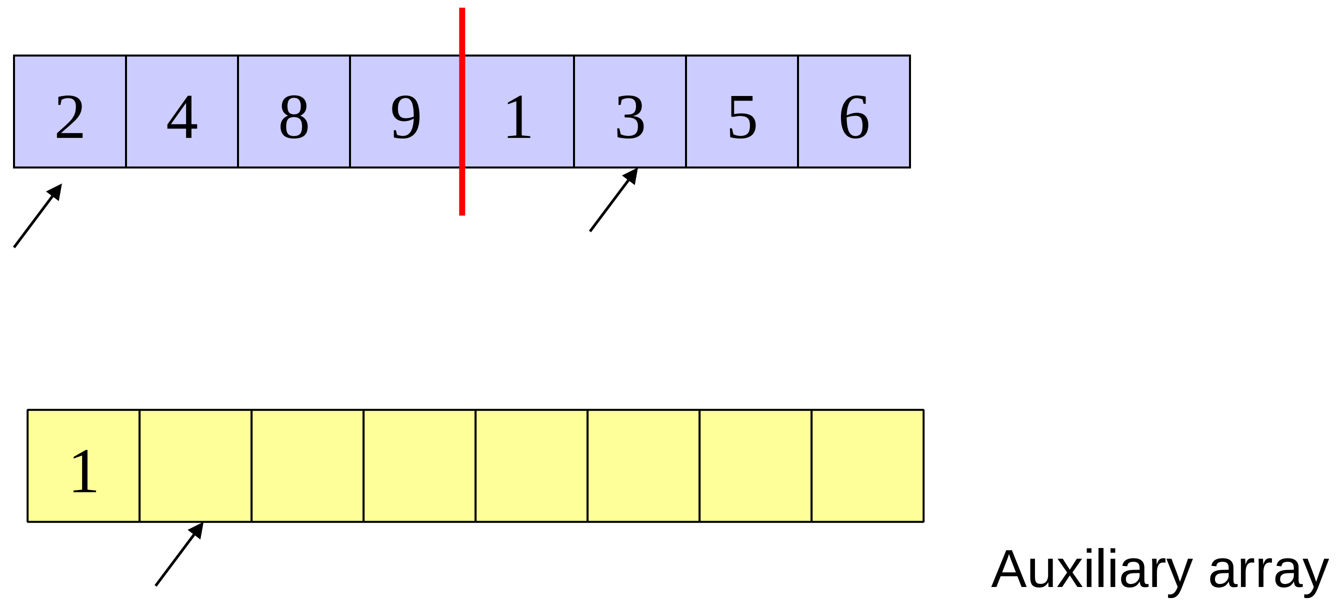
- The merging requires an auxiliary array.



Auxiliary array

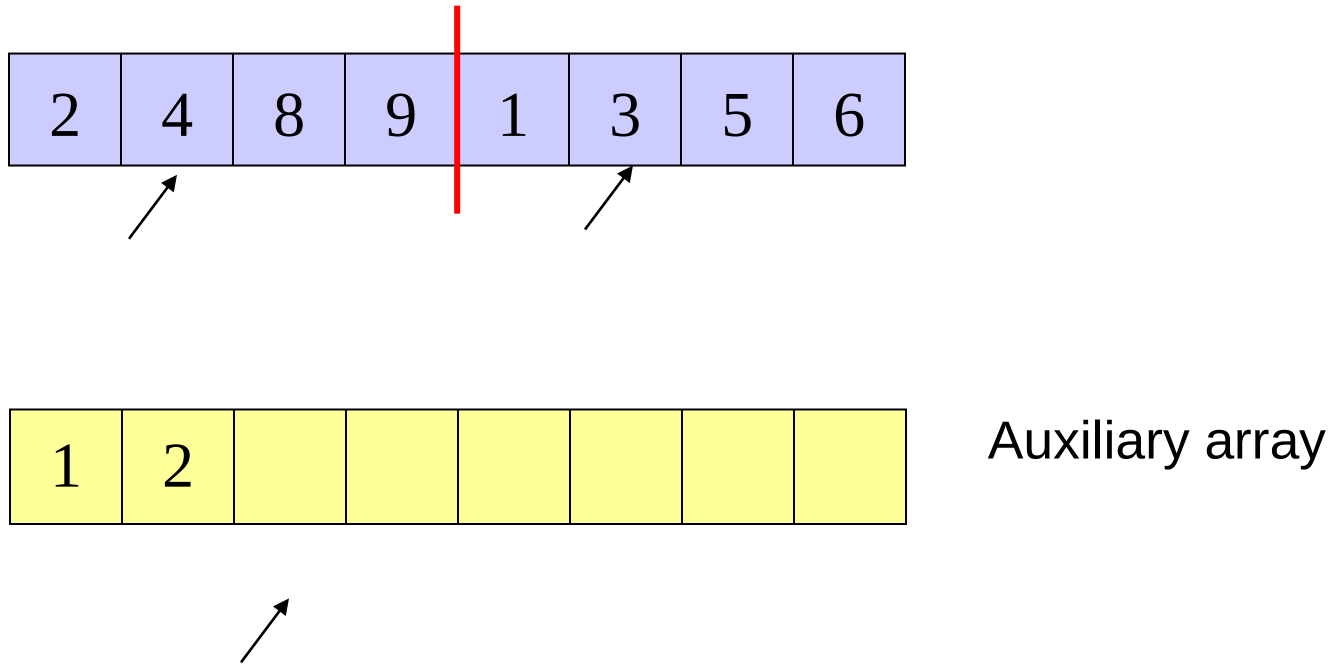
Working of Merge

- The merging requires an auxiliary array.



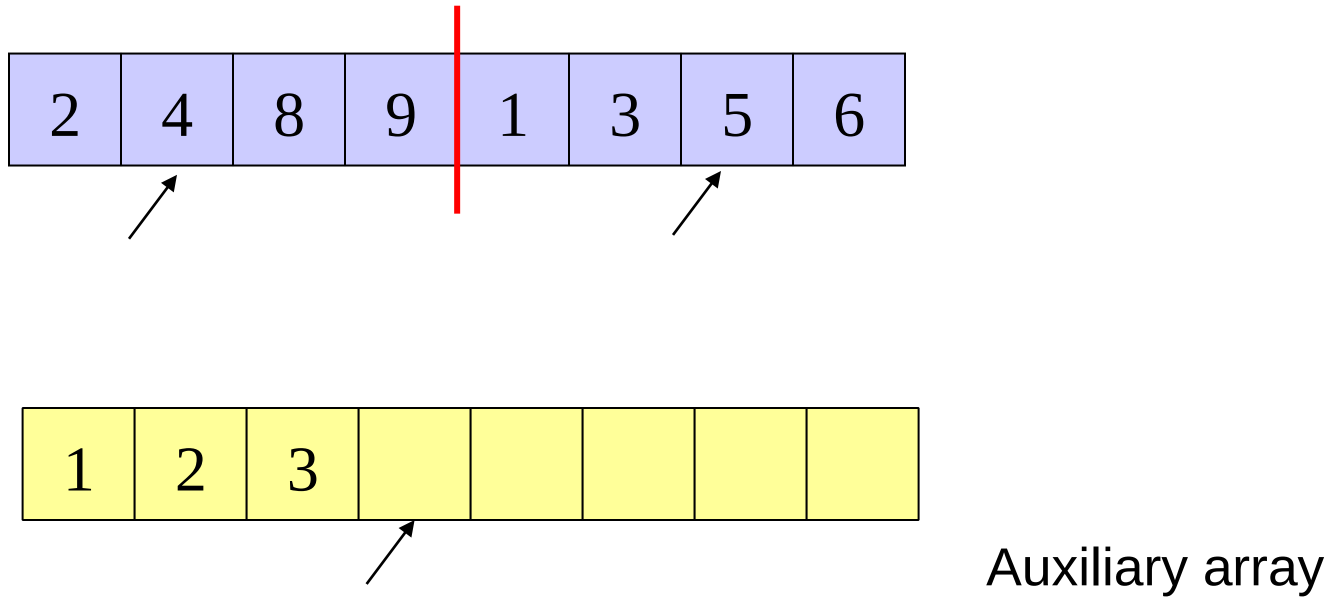
Working of Merge

- The merging requires an auxiliary array.



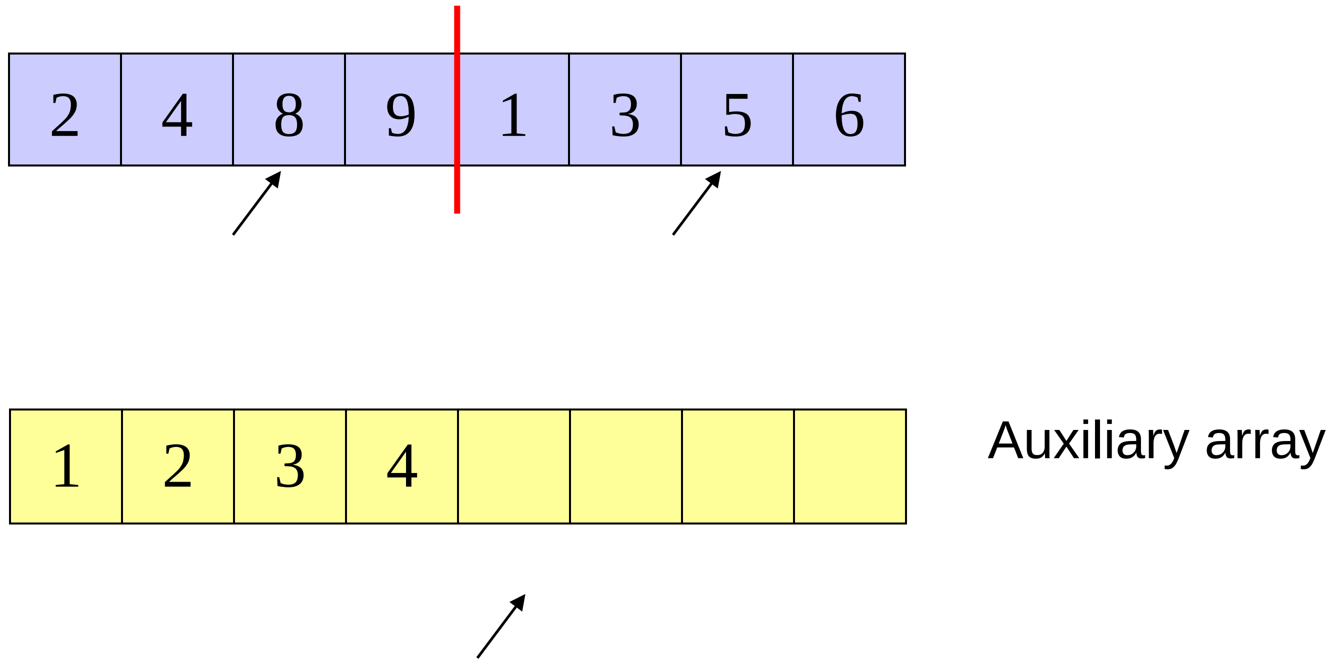
Working of Merge

- The merging requires an auxiliary array.



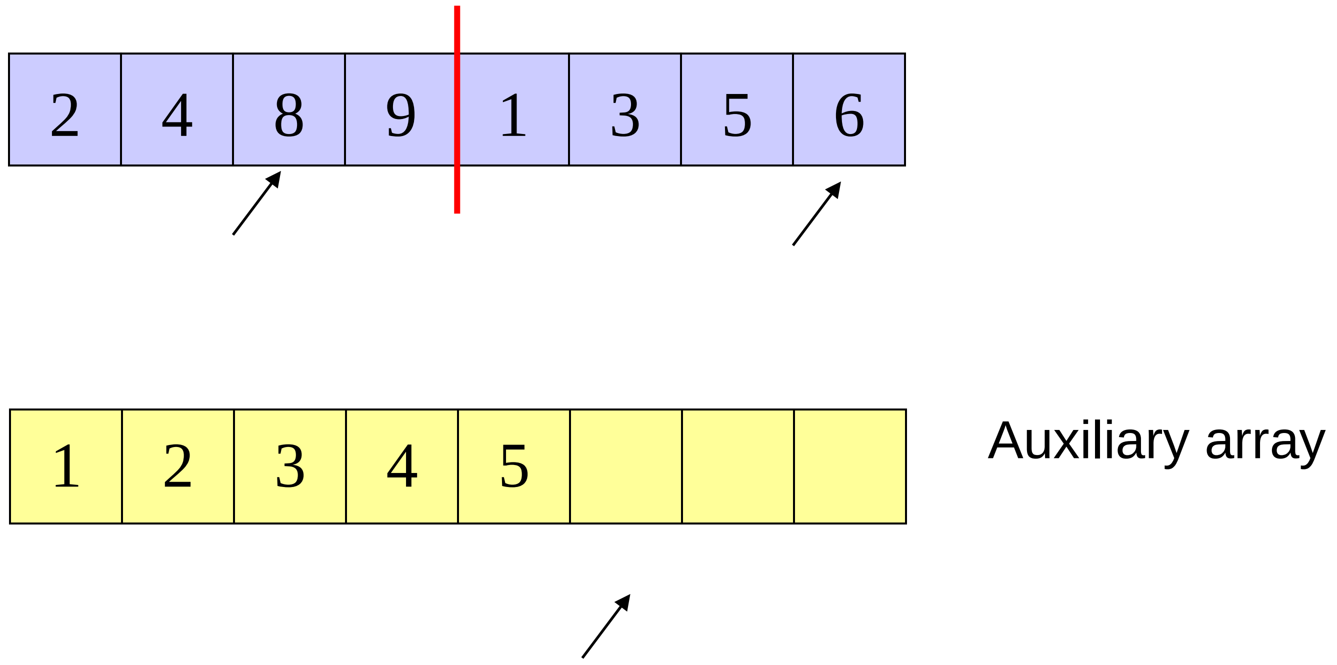
Working of Merge

- The merging requires an auxiliary array.



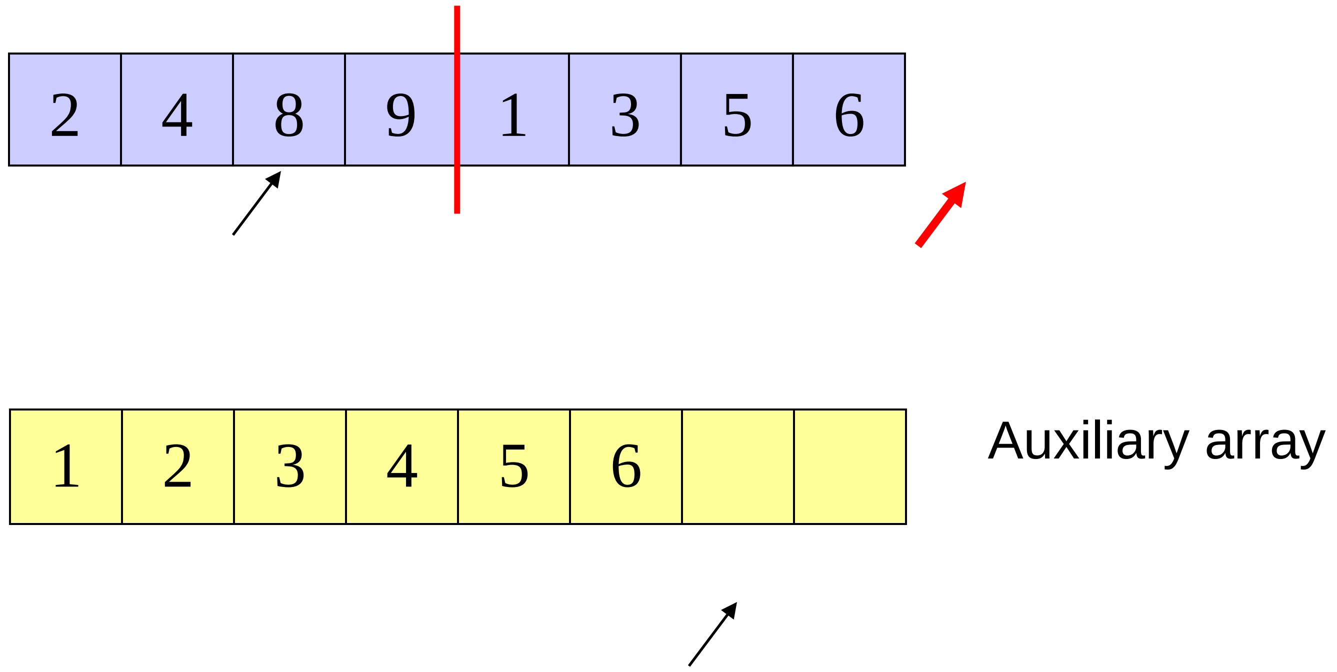
Working of Merge

- The merging requires an auxiliary array.



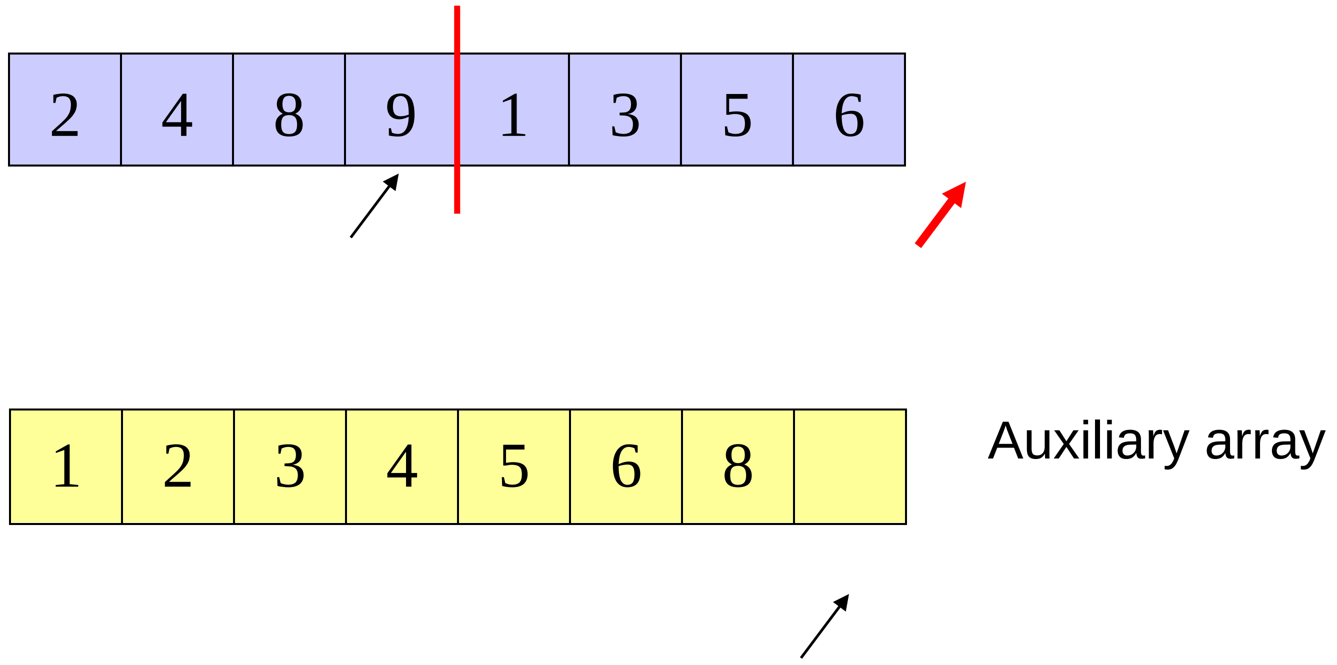
Working of Merge

- The merging requires an auxiliary array.



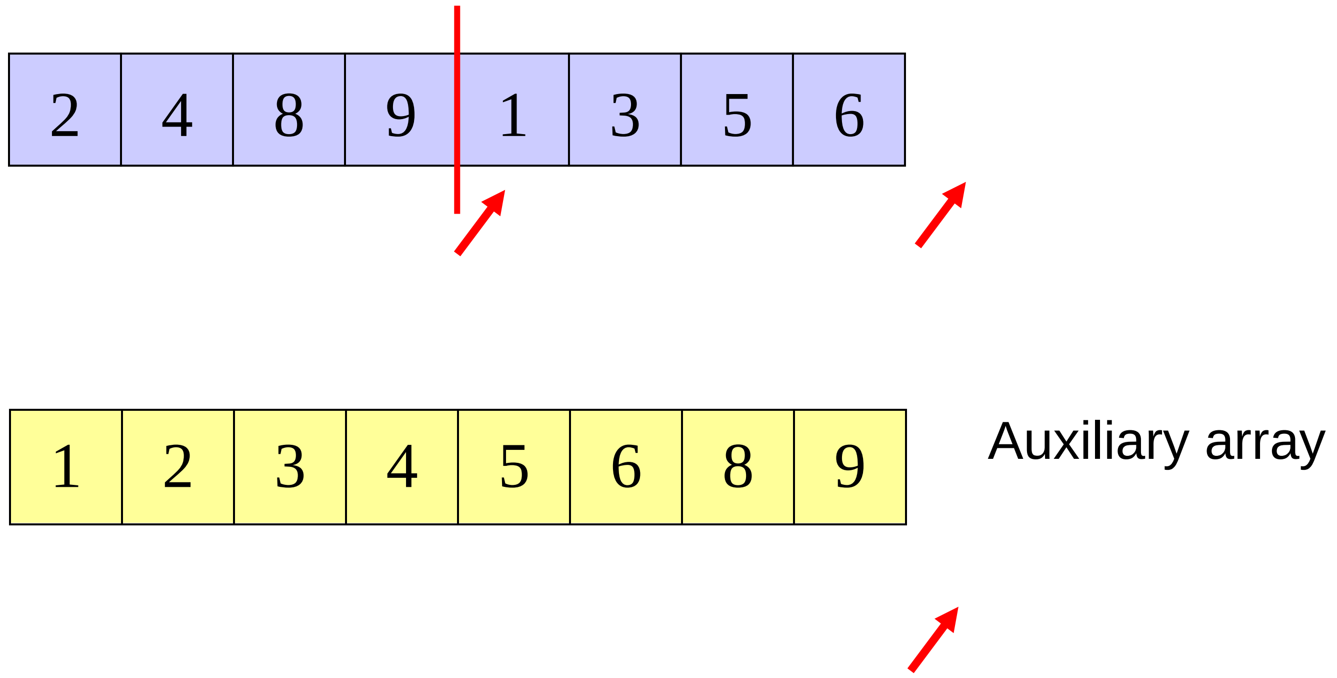
Working of Merge

- The merging requires an auxiliary array.



Working of Merge

- The merging requires an auxiliary array.



Merge Sort

- Lets Trace the algorithm

```
Algorithm MergeSort(low, high)
{ if (low<high) then
        { mid= $\lfloor (low + high)/2 \rfloor$
        MergeSort(low, mid)
        MergeSort(mid+1, high)
            Merge(low, mid, high)
        }
    }
```

Find the tracing in the video

Merge Sort -Analysis

- The recurrence relation for Merge sort is given by

$$T(n) = \begin{cases} a & n = 1, a \text{ is a constant} \\ 2T\left(\frac{n}{2}\right) + cn & n > 1, c \text{ is a constant} \end{cases}$$

- **Solution**

In the given relation a=2, b=2, f(n)=cn , n is power of b so $n=b^k$, $n=2^k$

T(n)=2T(n/2) +cn substitute T(n/2)=2T(n/4)+c(n/2)

= 2[2[T(n/4)+cn/2)] +cn

=4T(n/4)+2cn substitute T(n/4)=2T(n/8)+c(n/4)

=4[2T(n/8)+cn/4)+2cn

=8T(n/8)+3cn

The general pattern ?

= $2^kT(n/2^k)$+kcn $n=2^k$, k=logn

=nT(1)+logncn

= n+cnlogn considering only leading term and ignoring constants we get **T(n)= Θ(nlogn)**

Merge Sort -Summary

Properties summarized

- Merge Sort is useful for sorting linked lists.
- Merge Sort is a stable sort which means that the same element in an array maintain their original positions with respect to each other.
- Overall time complexity of Merge sort is Θ(nlogn).
  - i.e. its best, worst and average case time complexity is Θ(nlogn).
- It is more efficient as it is in worst case also the runtime is Θ(nlogn)

- The space complexity of Merge sort is O(n). This means that this algorithm takes a lot of space and may slower down operations for the large data sets.
- Merge sort is not in-place sorting

Quick Sort

- Quicksort is the other important sorting algorithm that is based on the ddivideand conquer approach.
- Unlike merge sort, which divides its input elements according to their position in the array, quicksort divides them according to their value.
- The idea of array partition is used in this sorting.
- A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

$$\underbrace{A[1] \ldots A[s-1]}_{\text{All are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n]}_{\text{All are} \ge A[s]}$$

Quick Sort

- Obviously, after a partition is achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of A[s] independently
- Now note the difference between the working of Merge sort and Quick sort
- In Merge sort the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions;
- In Quick sort, the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

Quick Sort

Pivot Element

- There are a number of ways to pick the pivot element.
- In this example, we will use the first element in the array:

5	3	1	9	8	2	4	7
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
5	3	1	9	8	2	4	7

Pivot

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

```
repeat
    Increment i   until a[i] ≥ p   [till u get greater number than pivot ]
    Decrement j until a[j] ≤ v  [till u get lesser number than pivot ] swap
        a[i] and a[j]
until i ≥ j swap (a[i],a[j]) swap(a[l],a[j]) return j
```

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7

p

Quick Sort

i j

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

```
repeat
    Increment i until a[i] ≥ p   [till u get greater number than pivot ]
    Decrement j until a[j] ≤ v  [till u get lesser number than pivot ] swap
        a[i] and a[j]
until i ≥ j swap (a[i],a[j]) swap(a[l],a[j]) return j
```

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7

p

Quick Sort

RNSIT-CSE Bengaluru

i **j**

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

```
repeat
    Increment i    until  a[i] ≥ p    [till u  get greater number than pivot ]
    Decrement j until  a[j] ≤ v  [till u  get lesser number than pivot ] swap
        a[i] and a[j]
until i ≥ j swap (a[i],a[j]) swap(a[l],a[j]) return j
```

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7

p ↗ ↗

Quick Sort

i j

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

Decrement **j** until **a[j] ≤ v** **[till u get lesser number than pivot]** swap **a[i]** and a**[j]**

until i ≥ j swap (a[i],a[j]) swap(a[l],a[j]) return j

	0	1	2	3	4	5	6	7
stop	5	3	1	9	8	2	4	7

p ↗ ↗

i j

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

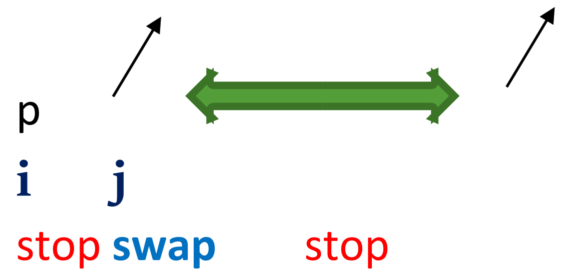
  Increment **i** until **a[i] ≥ p [till u get greater number than pivot]**

  Decrement **j** until **a[j] ≤ v [till u get lesser number than pivot]** swap

    a[i] and a[**j**]

until i ≥ j swap (a[i],a[j]) swap(a[l],a[j]) return j

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
5	3	1	9	8	2	4	7



Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Quick Sort

Following are the rules

repeat

 Increment i until **a[i] ≥ p [till u get greater number than pivot]**

 Decrement j until **a[j] ≤ v [till u get lesser number than pivot]** swap

  a[i] and a[**j]**

until i ≥ j swap (a[i],a[j]) swap(a[l],a[j]) return j

p

i **j**

Let the pivot

0	1	2	3	4	5	6	7
5	3	1	4	8	2	9	7

element be p, i.e.,

p=a[l], i=l, j=r+1

Following are the rules

Quick Sort

```
repeat
    Increment i    until  a[i] ≥  p    [till u  get greater number than pivot ]
    Decrement j until  a[j] ≤ v [till u  get lesser number than pivot ]
        swap  a[i] and a[j]
until i ≥ j swap
(a[i],a[j])
swap(a[l],a[j])
return j
```

0	1	2	3	4	5	6	7
5	3	1	4	8	2	9	7

Quick Sort

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

```
repeat
    Increment i   until  a[i] ≥ p    [till u  get greater number than pivot ]
    Decrement j until  a[j] ≤ v   [till u  get lesser number than pivot ]
        swap  a[i] and a[j]
until i ≥ j swap
(a[i],a[j])
```

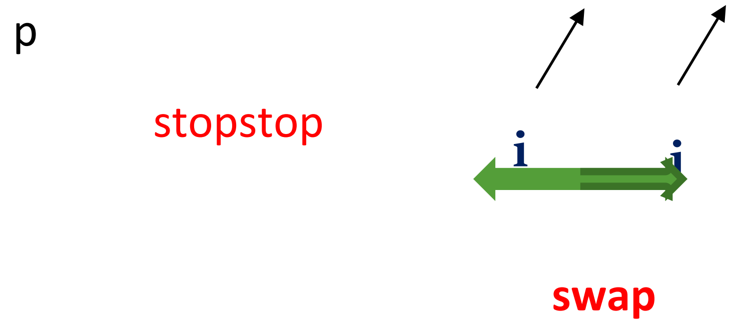
0	1	2	3	4	5	6	7
5	3	1	4	8	2	9	7

⟸

Quick Sort

swap(a[l],a[j])

return j



Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

repeat

Increment **i** until **a[i] ≥ p** **[till u get greater number than pivot]**

Quick Sort

Decrement **j** until **a[j] ≤ v [till u get lesser number than pivot]**

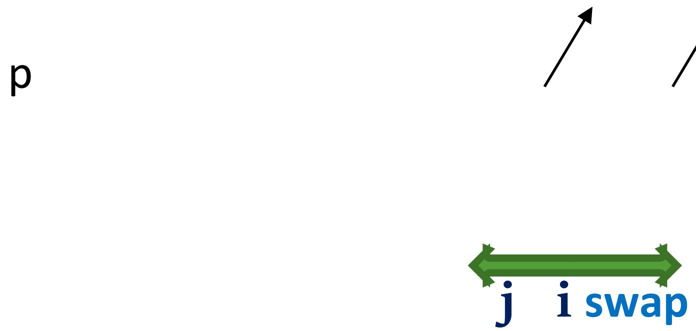
swap **a[i]** and a[j]

until i ≥ j swap (a[i],a[j])

0	1	2	3	4	5	6	7
5	3	1	4	2	8	9	7

swap(a[l],a[j])

return j



Quick Sort

i **j**

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

```
repeat
    Increment i  until  a[i] ≥ p   [till u get greater number than pivot ]
    Decrement j until  a[j] ≤ v  [till u get lesser number than pivot ]
        swap  a[i] and a[j]
until i ≥ j swap
(a[i],a[j])
```

0	1	2	3	4	5	6	7
5	3	1	4	2	8	9	7

Quick Sort

```
swap(a[l],a[j])
return j
```

p

stop

j i swap

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

```
repeat
    Increment i   until a[i] ≥ p   [till u get greater number than pivot ]
    Decrement j until a[j] ≤ v  [till u get lesser number than pivot ]
        swap a[i] and a[j]
until i ≥ j swap (a[i],a[j])
```

0	1	2	3	4	5	6	7
5	3	1	4	8	2	9	7

j i swap

Quick Sort

```
swap(a[l],a[j])
return j
```

p

stop

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

```
repeat
    Increment i    until  a[i] ≥ p    [till u  get greater number than pivot ]
    Decrement j until  a[j] ≤ v  [till u  get lesser number than pivot ]
        swap  a[i] and a[j]
```

j i swap

Quick Sort

until i ≥ j swap (a[i],a[j])

0	1	2	3	4	5	6	7
5	3	1	4	2	8	9	7

swap(a[l],a[j])

return j

p

stop

Let the pivot element be p, i.e., **p=a[l], i=l, j=r+1**

Following are the rules

j **i**

Quick Sort

```
repeat
    Increment i   until  a[i] ≥  p    [till u  get greater number than pivot ]
    Decrement j until  a[j] ≤ v  [till u  get lesser number than pivot ]
        swap  a[i] and a[j]
until i ≥ j swap
(a[i],a[j])
swap(a[l],a[j])
return j
```

0	1	2	3	4	5	6	7
5	3	1	4	2	8	9	7

Quick Sort

p $\nearrow$ $\nearrow$ **swap**

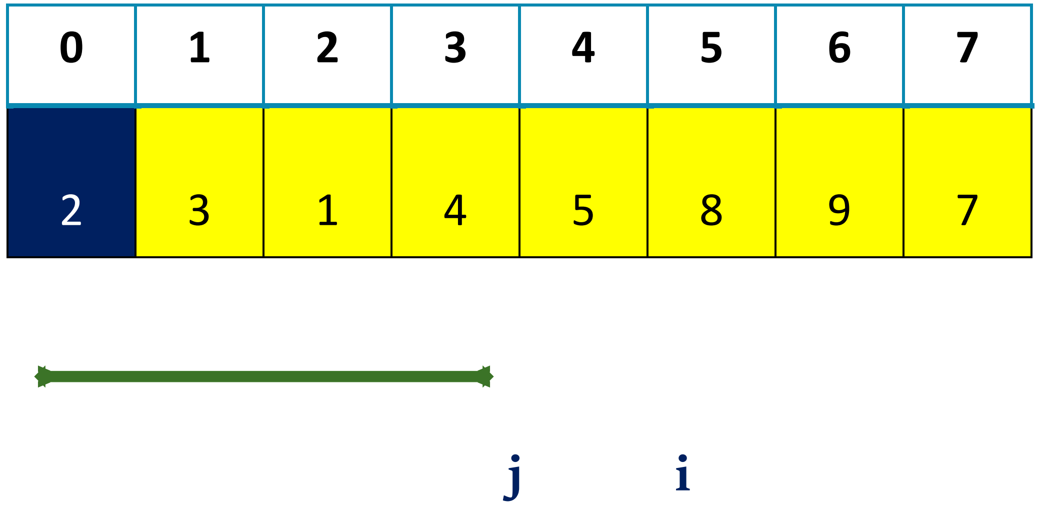
j **i**

Quick Sort

Let the pivot element be p, i.e., **p=a[l], i=l, j= r+1**

Following are the rules

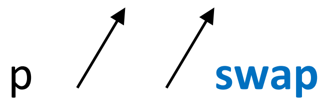
```
repeat
    Increment i   until  a[i] ≥ p   [till u get greater number than pivot ]
    Decrement j until  a[j] ≤ v [till u get lesser number than pivot ]
        swap  a[i] and a[j]
until i ≥ j swap
(a[i],a[j])
```



0	1	2	3	4	5	6	7
2	3	1	4	5	8	9	7

Quick Sort

```
swap(a[l],a[j])
return j
```



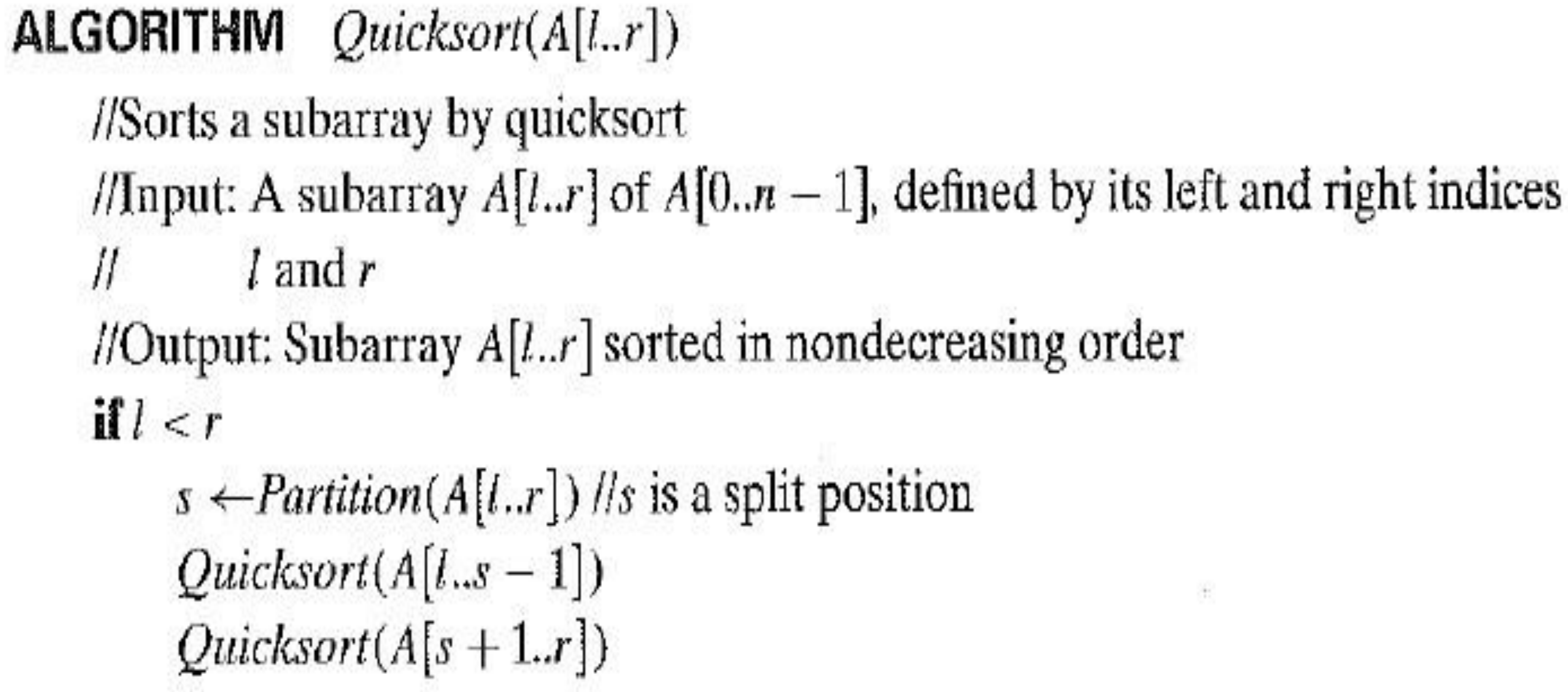
Do it yourself

- Obtain the first partition for the following set of elements considering the first element as the pivot element

  65, 70, 75, 80, 85, 60, 55, 50, 45

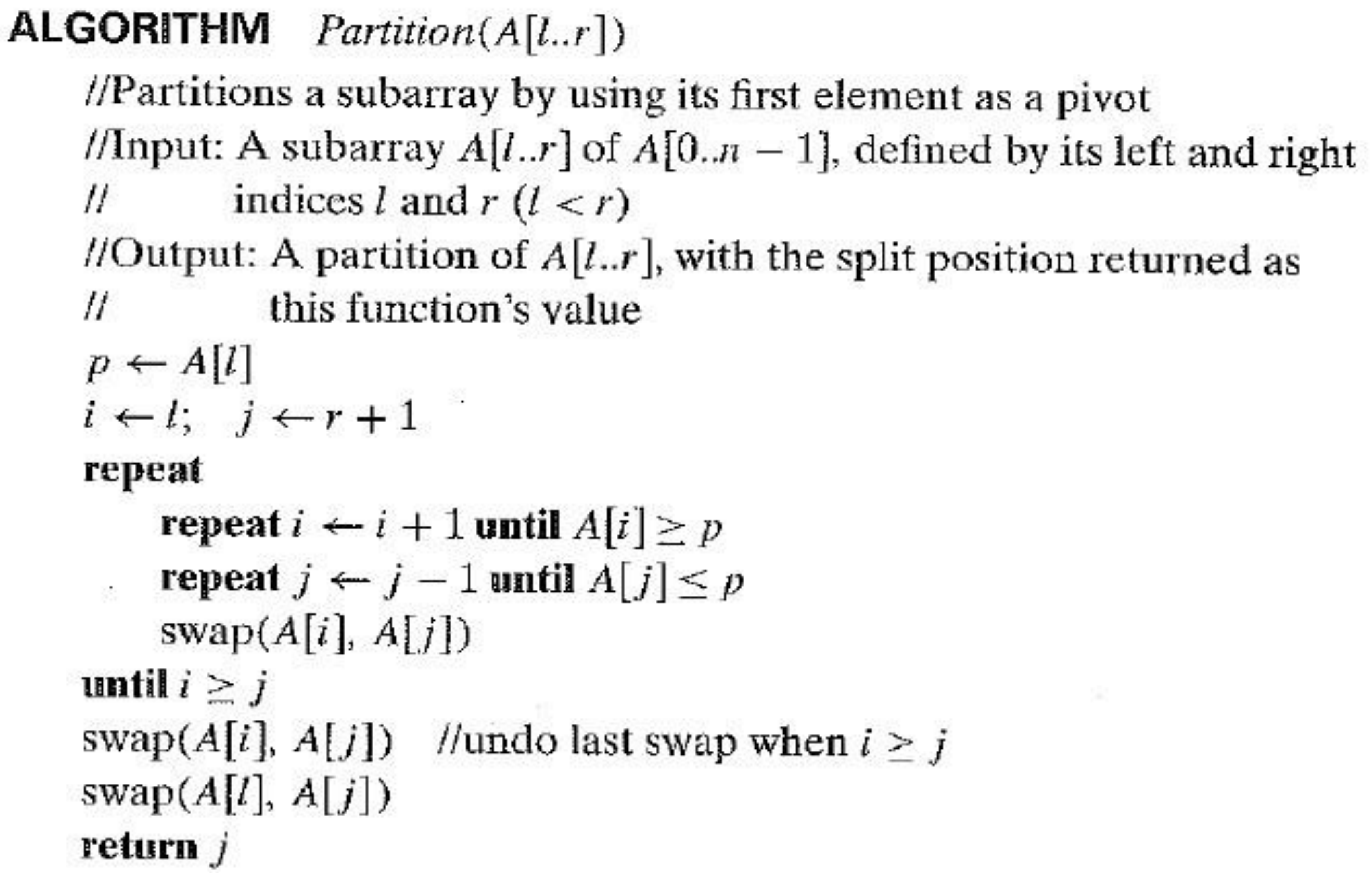
- Apply quicksort to sort the list **E, X, A,M, P, L, E** in alphabetical order

Quick Sort



```
ALGORITHM  $Quicksort(A[l..r])$
    //Sorts a subarray by quicksort
    //Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right indices
    //        $l$ and $r$
    //Output: Subarray $A[l..r]$ sorted in nondecreasing order
    if $l < r$
        $s \leftarrow Partition(A[l..r])$ //$s$ is a split position
        $Quicksort(A[l..s-1])$
        $Quicksort(A[s+1..r])$
```

Quick Sort



```
ALGORITHM  Partition(A[l..r])
    //Partitions a subarray by using its first element as a pivot
    //Input: A subarray A[l..r] of A[0..n − 1], defined by its left and right
    //        indices l and r (l < r)
    //Output: A partition of A[l..r], with the split position returned as
    //          this function's value
    $p \leftarrow A[l]$
    $i \leftarrow l$;   $j \leftarrow r + 1$
    repeat
        repeat $i \leftarrow i + 1$ until $A[i] \geq p$
        repeat $j \leftarrow j - 1$ until $A[j] \leq p$
        swap($A[i]$, $A[j]$)
    until $i \geq j$
    swap($A[i]$, $A[j]$)   //undo last swap when $i \geq j$
    swap($A[l]$, $A[j]$)
    return $j$
```

Quick Sort –Analysis (Best Case)

- The recurrence relation is given by

$$T(n) = \begin{cases} a & n = 1, a \text{ is a constant} \\ 2T\left(\frac{n}{2}\right) + n & n > 1\text{ , } c \text{ is a constant} \end{cases}$$

- **Solution**

In the given relation a=2, b=2, f(n)=cn , n is power of b so $n=b^k$, $n=2^k$

T(n)=2T(n/2) +n substitute T(n/2)=2T(n/4)+(n/2)

= 2[2[T(n/4)+n/2)] +n

=4T(n/4)+2n substitute T(n/4)=2T(n/8)+(n/4)

=4[2T(n/8)+n/4)+2n

=8T(n/8)+3n

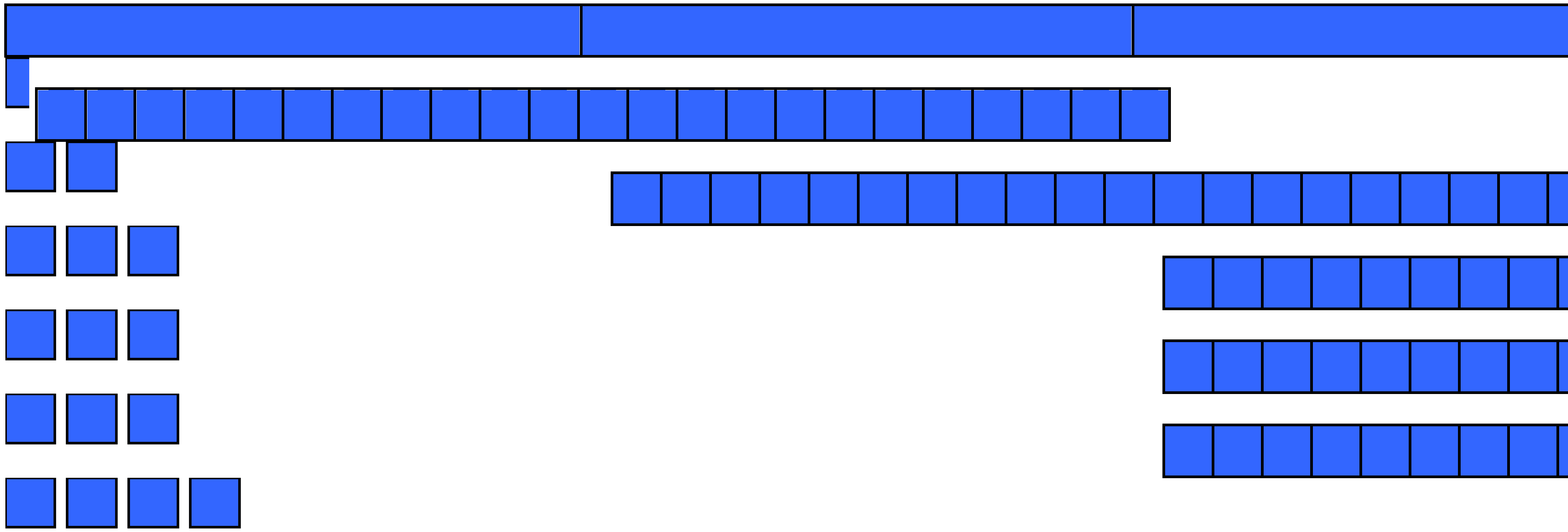
The general pattern ?

= $2^k T(n/2^k)$+kn $n=2^k$, k=logn

=nT(1)+logn n

= n+cnlogn considering only leading term and ignoring constants we get **T(n)** $_{\text{Best}}$ **= Θ(nlogn)**

Quick Sort- Analysis (Worst Case)



Quick Sort- Analysis (Worst Case)

- The recurrence relation for worst case analysis is given by

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

RNSIT-CSE Bengaluru

Average case

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

$$C_{avg}(n) \approx 2n \ln n \approx 1.38n \log_2 n.$$

ESTD:2001
An Institute with a Difference

Design and Analysis of Algorithms

Divide and Conquer

Manjula L
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

Strassen’s Matrix Multiplication

- Let A and B be two n x n matrices
- The product matrix C= AB is also an n x n matrix whose i, jth element is formed by taking the elements in the ith row of **A**and jth column of **B** and multiplying them to get
- $C(i, j) = \sum_{1\leq k\leq n} A(i, k)B(k, j)$ for all i and j between 1 and n
- To compute C(i, j) using the formula above how many multiplications are needed?
- Consider an example

21 ▪ Where, 22 21 22 21 22

$c_{11}=a_{11}b_{11}+a_{12}b_{21}$

Time complexity ? **$\Theta(n^3)$**

$a_{11}\ a_{12}$B=$b_{11}\ b_{12}$then C= $c_{11}\ c_{12}$ A=

a ab bc c

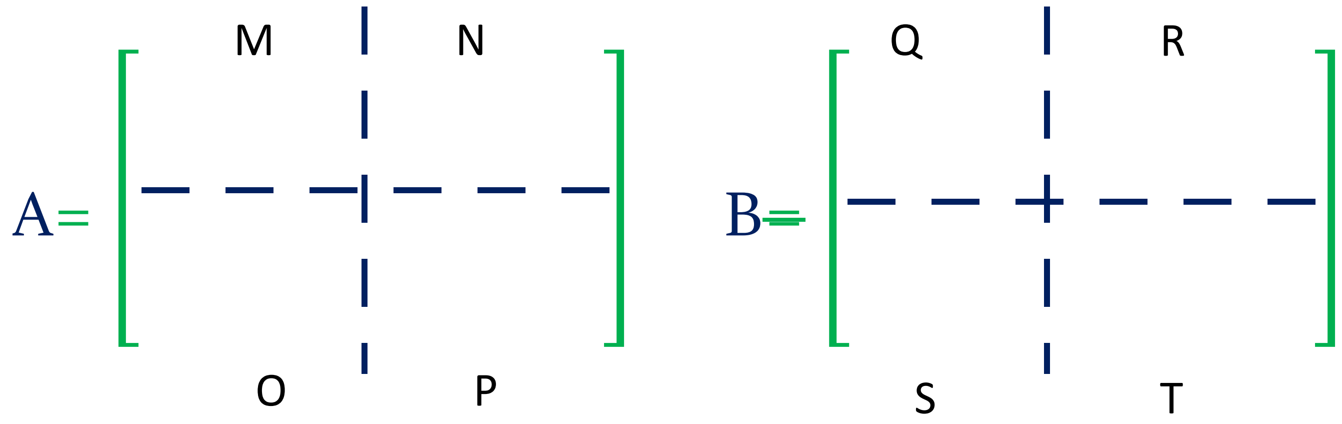
$c_{12}=a_{11}b_{12}+a_{12}b_{22}$ 8 multiplications

$c_{21}=a_{21}b_{11}+a_{22}b_{21}$ $c_{22}=a_{21}b_{12}+a_{22}b_{22}$

Strassen’s Matrix Multiplication

- Can we use **Divide and Conquer** approach to multiply two **n x n** matrices ?
- Let's assume that **n** is power of **2**, i.e., there exists a non-negative constant **k** such that **$n=2^k$**

- If **n** is not power of **2** then add enough rows and columns of **zeros** to both **A** and **B** so that the resultant dimensions are power of **2**.
- Here is the application of Divide and Conquer approach



$$APrep = \left[\begin{array}{cc|cc} 0 & 8 & 0 & 0 \\ 10 & 1 & 0 & 0 \\ \hline 8 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

$$BPrep = \left[\begin{array}{cc|cc} 9 & 6 & 4 & 8 \\ -3 & 2 & 7 & 1 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

$$a = \begin{bmatrix} 0 & 8 \\ 10 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad c = \begin{bmatrix} 8 & 6 \\ 0 & 0 \end{bmatrix} \quad d = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$e = \begin{bmatrix} 9 & 6 \\ -3 & 2 \end{bmatrix} \quad f = \begin{bmatrix} 4 & 8 \\ 7 & 1 \end{bmatrix} \quad g = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad h = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Strassen’s Matrix Multiplication

- Consider the following situation

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

- Then

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned}$$

- Where

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

RNSIT-CSE Bengaluru

- Consider the following matrices and compute the product using Strassen’s Method
- A= $\begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix}$ B= $\begin{bmatrix} 1 & 3 \\ 4 & 7 \end{bmatrix}$

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned}$$

P= (2+5) (1+7)= 7* 8= 56

Q= (3+5)* 1= 8

R= 2* (3-7)= -8

S= 5*(4-1)= 15

T= (2+4) *7= 42

U= (3-2)* (1+3)= 4

V= (4-5)* (4+7)=-11

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

C11 = 56+ 15-42-11= 18

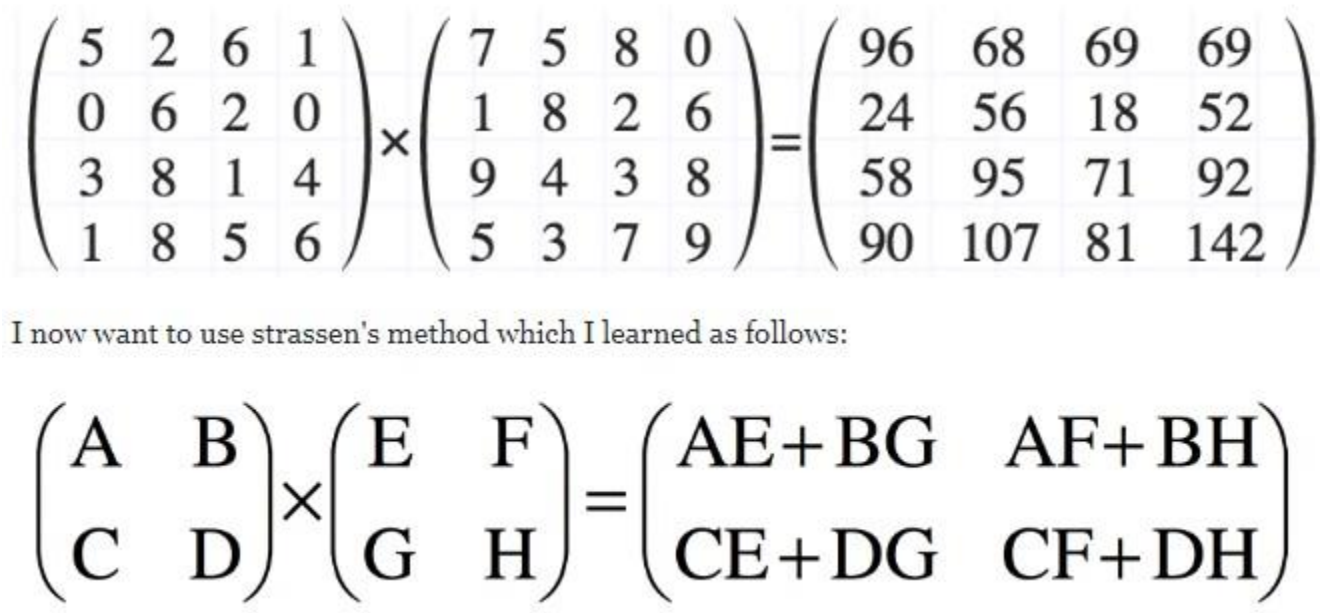
C12= -8+ 42= 34

C21= 8+15 = 23

C22= 56-8 -8+4 = 44

Strassen’s Matrix Multiplication

Strassen's 4X4 Matrix Multiplication



$$\begin{pmatrix} 5 & 2 & 6 & 1 \\ 0 & 6 & 2 & 0 \\ 3 & 8 & 1 & 4 \\ 1 & 8 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 7 & 5 & 8 & 0 \\ 1 & 8 & 2 & 6 \\ 9 & 4 & 3 & 8 \\ 5 & 3 & 7 & 9 \end{pmatrix} = \begin{pmatrix} 96 & 68 & 69 & 69 \\ 24 & 56 & 18 & 52 \\ 58 & 95 & 71 & 92 \\ 90 & 107 & 81 & 142 \end{pmatrix}$$

I now want to use strassen's method which I learned as follows:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$$

Time efficiency

- M(n)= 7 M(n/2) for n>1, M(1)=1 ▪

Since n= 2 ^ k

- M(2^ K)= 7 M (2^ (k-1))

- 7 ^ i M(2 ^ (k-k)
- 7^k
- K=log2 n

- $n ^ 2.807$

Pros of Divide and Conquer Strategy

- Solving difficult problems
- Algorithm efficiency
- Parallelism – Suitable for multiprocessor machines
- Memory access - *optimal* cache-oblivious algorithms

Cons of Divide and Conquer Strategy

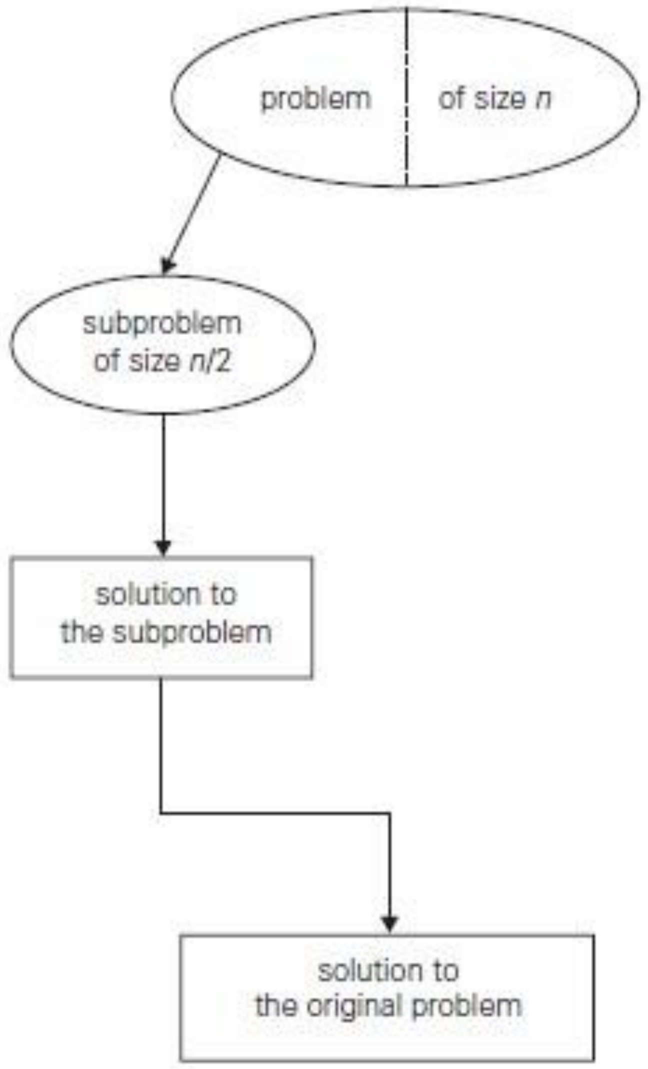
- **Divide and Conquer** strategy **uses** recursion that makes it a little slower and if a little error occurs in the code the program may enter into an infinite loop.
- Usage of explicit stacks may make **use** of extra space.

Decrease and Conquer

- This technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. Once such relationship is established, it can be exploited either top down (recursively) or bottom up (without a recursion).

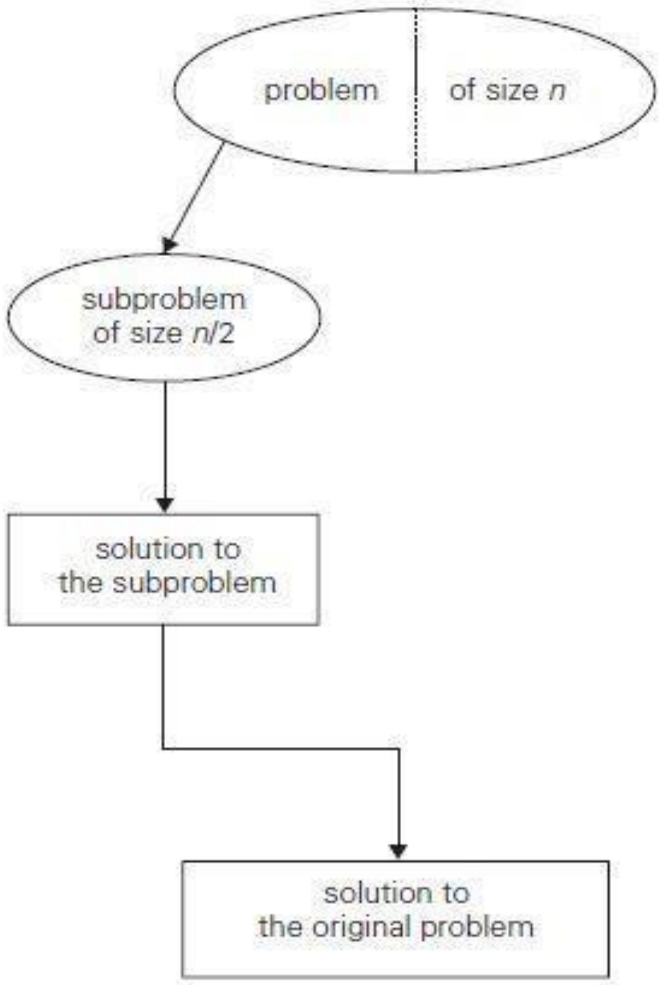
- There are three major variations of decrease-and-conquer:
  - Decrease by a constant.
  - Decrease by a constant factor.
  - Variable size decrease.

Decrease by a constant



$$f(n) = \begin{cases} f(n-1)\cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases}$$

Decrease by a constant Factor



$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$

Variable size decrease

- gcd (m, n) = gcd (n, m mod n).

THANK

RNSIT-CSE
Bengaluru

YOU