

Design and Analysis of Algorithms

Module 1

Algorithm

- An algorithm is a tool for solving a given problem



- An algorithm is a finite sequence of instructions such that each instruction:
 - Has a well defined and clear meaning.
 - Can be performed in a finite amount of time and
 - Can be performed easily.

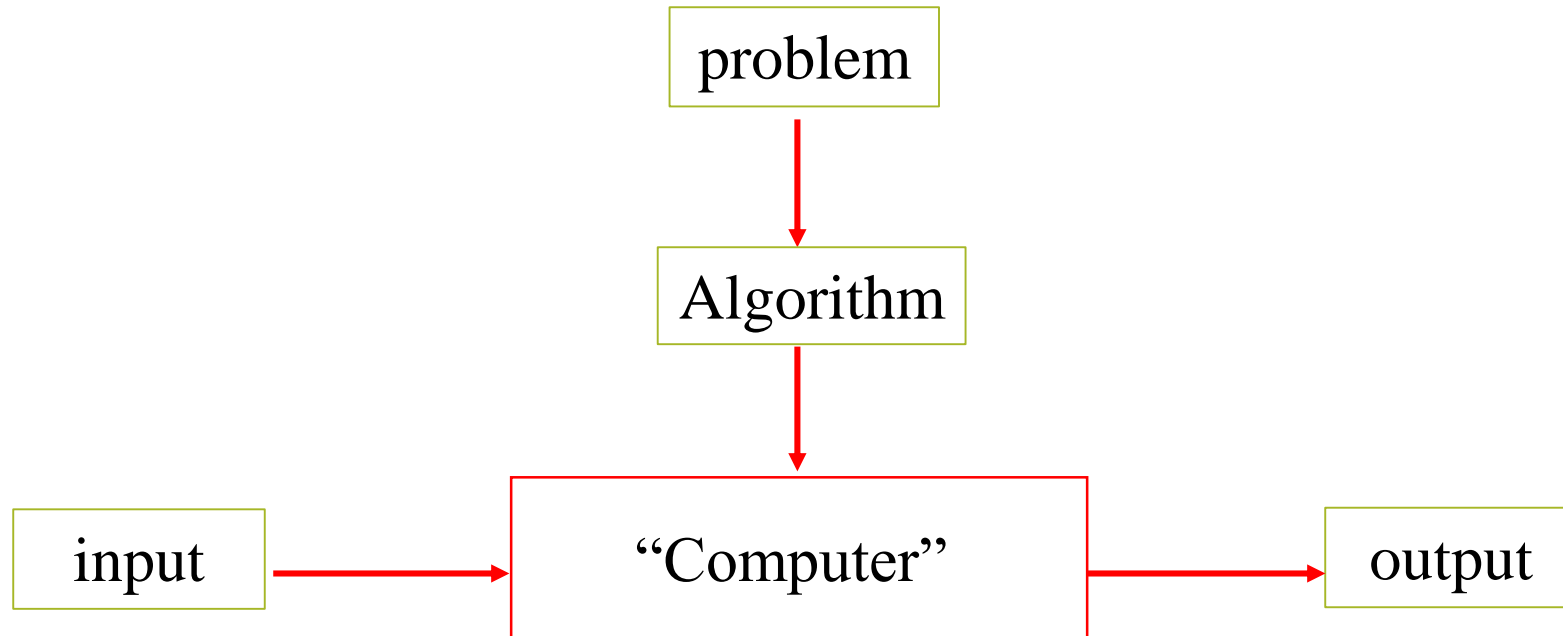
Algorithm

- An algorithm is composed of a finite set of steps, each of which may require one or more operations.
- The possibility of a computer carrying out these operations necessitates that certain constraints be placed on the type of operations an algorithm can include.

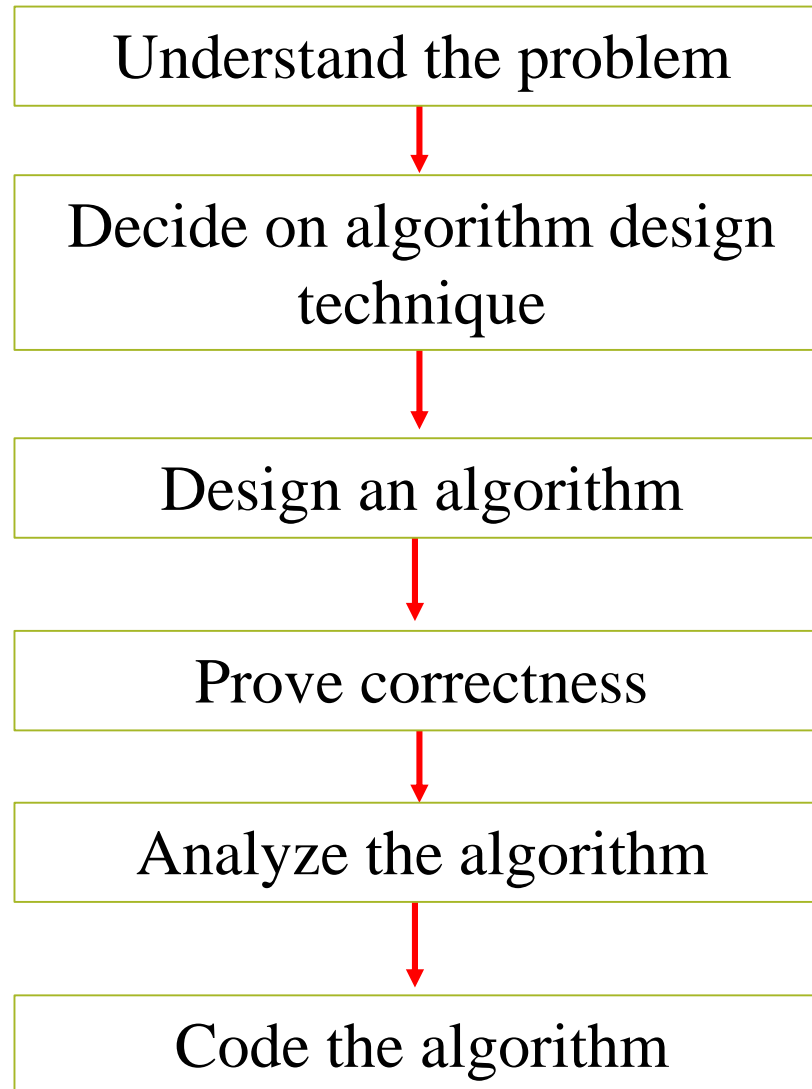
Definition

- An *Algorithm* is a finite set of instructions that accomplishes a particular task.
- An algorithm is a logically-arranged sequence of a finite number of instructions which when implemented solves the given problem.
- 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.

Notion of algorithm



Fundamentals of algorithmic problem solving



Properties of Algorithm

- 1. Input :** An algorithm must be supplied with zero or some finite input values externally from a specified set in order to solve the computational problem and generate some output. The input data is transformed during the computation in order to produce the output.
- 2. Output :** The algorithm produces some finite set of outputs after applying some operations on the given set of input values. The output values are the solution. The output can be anything from data returned to the calling algorithm, displaying the message, printing the calculation, etc. It is possible to have no output.
- 3. Finiteness:** The algorithm must be terminated after executing a finite number of steps. So an algorithm must be a well-defined, ordered set of instructions.

Properties of Algorithm

4. Definiteness: Each step of an algorithm must be clear and unambiguous so that the actions can be carried out without any ambiguity. For example, same symbol should not be used to mean multiplication as well as division at two different places in the algorithm.

5. Effectiveness: The algorithm must perform each step correctly and in a finite amount of time, therefore time tends to be more important in calculating the effectiveness of an algorithm. The space and other resources taken up by algorithm also play vital role in effectiveness of an algorithm. Effectiveness is precisely measured after translating the algorithm into a computer program.

6. Correctness: An algorithm must produce the correct output values for all legal input instances of the problem.

Properties of Algorithm

- 7. Generality:** The algorithm should be applicable to all problems of a similar form.
- 8. Multiple views:** Same algorithm may be represented in different ways.
- 9. Multiple availability:** Several algorithms for solving the same problem may exist, with different properties.

Expressing Algorithms

1. Pseudocode

- Pseudocode is one of the preferred notations for describing algorithms. Algorithms are expressed in an English-like language called pseudocode that is designed by using basic programming constructs, such as sequence, branching and looping. So we can say that pseudocode is a combination of English and programming constructs.
- It focuses on the fundamental operation of the program instead of program design issues.
- There is no standard for pseudocode.

❖ Write a pseudocode for searching the smallest element from the list of given elements

```
1.  Min_Array (A, N)
2.  Input: An array 'A' with 'N' elements
    is given
3.  Output: Display the minimum
    element
4.  Start
5.  Set Min = A[0]
6.  for (i = 1 to N-1) do
    1.  If (A[i] < Min) then
        1.  Set Min = A[i]
7.  Print (Min)
8.  Stop
```

```
1.  Algorithm Min_Array (A, N)
2.  // Input: An array 'A' with 'N' elements
    is given
3.  // Output: Display the minimum
    element
4.  {
5.      Min = A[0];
6.      for i := 1 to N-1 do
        1.  If (A[i] < Min) then
            1.  Min := A[i];
7.      return Min;
8.  }
```

2. Flow chart

The algorithms are graphically represented using flowcharts.



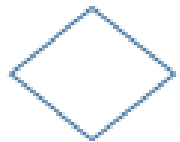
Ellipse: To represent the start and end of algorithms



Parallelogram: To represent the input and output or the read and write operations of algorithms



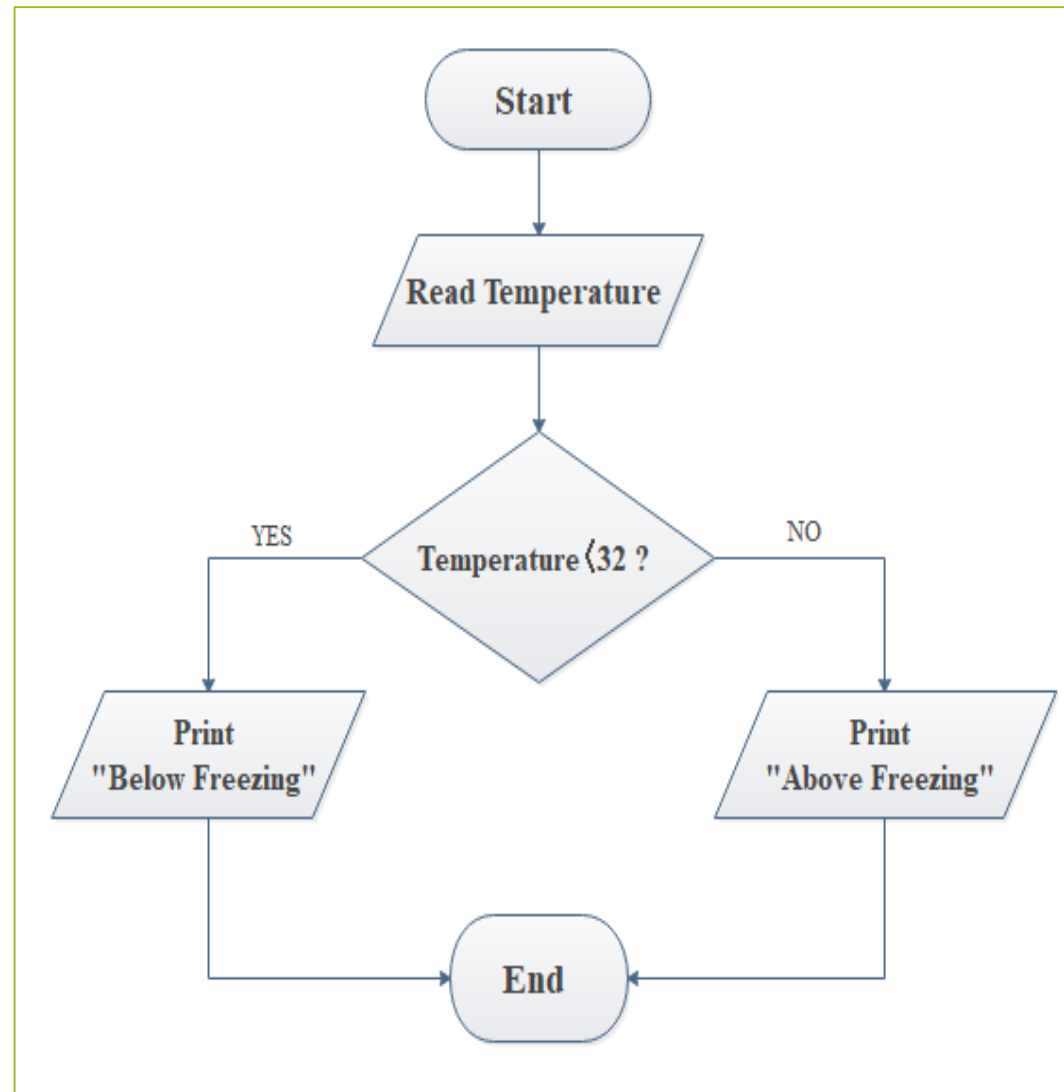
Rectangle: To represent the processing of instructions



Rhombus: To represent branching of the statements: there will be one entry point and more than one exit point

Arrow: To represent the flow of data or the sequence of statements

Example



Performance Analysis of Algorithms

- The execution of an algorithm requires various resources of the computer system to complete the task.
- The **important resources** which contribute to the **efficiency of algorithms** are the **memory space** and the **time required for successful execution** of an algorithm.
- The **goal** for performance analysis is to **determine the resources needed** for the execution of an algorithm.

Performance Analysis of Algorithms

Analysis of algorithms is divided into the following two categories:

1. **Space complexity:** The *space complexity* of an algorithm is the amount of memory it needs to run/execute to completion.
2. **Time complexity:** The *time complexity* of an algorithm is the amount of computer time it needs to run to completion.

Performance Analysis of Algorithms : Space complexity

- The space is needed for :
 - ❖ storing the variables, constants and identifiers
 - ❖ Program instruction codes
 - ❖ Function calls
 - ❖ Return value, etc.
- The space complexity depends on the static and dynamic allocation of memory space during the **compile time** and **run time** respectively.

Performance Analysis of Algorithms : Space complexity

- The space needed by the algorithm is seen to be the sum of the following components:
- A **fixed part** that is independent of the characteristics (e.g., number, size) of the inputs and outputs. This part typically includes
 - Instruction space (the space for the code),
 - space for simple variables and fixed-size component variables,
 - space for constants, and so on
 - *is allocated during the compile time.*

Performance Analysis of Algorithms : Space complexity

- A **variable part** that consists of the space needed by
 - component variables whose size is dependent on the particular problem instance being solved,
 - the space needed by referenced variables,
 - formal parameters, local variables and return address in the stack in case of recursion,
 - variables, constants and identifiers
 - *declared using dynamic declaration, is allocated during the **run time**.*

Performance Analysis of Algorithms : Space complexity

- Total space complexity can be defined as
 - $S(A) = S(C) + S(R)$
 - Where $S(A)$ is the space required to run the algorithm, $S(C)$ is the space required at compile time which remains fixed and $S(R)$ is the space required at run time which is not fixed.
- The space requirement/complexity $S(P)$ of any algorithm P may as
 - **$S(P) = c + Sp(\text{instance characteristics})$, where c is a constant.**
 - Where c is the fixed space and
 - $Sp(\text{instance characteristics})$ represents the variable space
- When analyzing the space complexity of an algorithm, we concentrate solely on estimating $Sp(\text{instance characteristics})$.

Space Complexity Calculation

- Let P be the algorithm
- Let $S(P)$ be the space requirement for algorithm
- Then

$$S(P) = C + S_p$$

- Where
 - C is the fixed space
 - S_p is the variable space
- When analyzing space complexity of an algorithm we concentrate solely on estimating S_p (variable space).

Performance Analysis of Algorithms : Space complexity

Example 1:

```
Algorithm abc(x,y,z)
    return x*y*z + (x-y)
```

$$S(P) = c + S_p$$

x=1 unit

y=1 unit

z=1 unit

$$S(P) = 1+1+1=3$$

$$S(P)= 3$$

Example 2:

```
Algorithm sum(x,n)
{
    total :=0
    for i → 1 to n do
        total:= total + x[i]
}
```

$$S(P) = c + S_p$$

x=1 unit

n=1 unit

total=1 unit

$$S(P)= 3+n$$

Performance Analysis of Algorithms : Time complexity

- Time complexity is the measurement of time required for the execution of an algorithm.
- The Time $T(P)$ taken by an algorithm P is the sum of the **compile time** and **run time**.
- The compile time does not depend on the instance characteristics.
- We may assume that a compiled program will be run several times without recompilation.
- Consequently, In time complexity we *consider only run time*. This run time is denoted by t_p (instance characteristics)

Performance Analysis of Algorithms : Time complexity

- The execution time is not really the physical time corresponding to the execution of the algorithms on a computer but it is **the number of machine instructions** which a program executes during its running time.
- The concept of execution time has been expressed as a function of problem size which is popularly known as time complexity of an algorithm.
- So the time complexity is the estimate of the number of basic operations executed by an algorithm.

Performance Analysis of Algorithms : Time complexity

- It is calculated by computing the cost of execution of each statement of the program multiplied by the number of times the statement is executed.so,
- if $T(n)$ is the execution time of an algorithm then

**Total cost of the statement = Cost of the statement executing once
* No. of times a statement is executed.**

$T(n)$ = sum of total cost of all the statements

Time Complexity

- Time complexity is sum of two components
- **Compile time**
 - Does not depend on instance characteristics.
 - Also once compiled program will be run several times without recompilation
- **Run time**
 - Dependent on particular problem instance
 - So time complexity is calculated as

$$T(P) = C + T_p$$

Where

$T(P)$ - time complexity of algorithm P

C - Compile time

T_p - Run Time

Time Complexity

- The number of steps any program statement is assigned depends on the kind of statement. For example,
 - comments count as zero steps,
 - an assignment statement which does not involve any calls to other algorithm is counted as one step,
 - for iterative statements such as the for, while and repeat-until statements, we consider the step counts only for the control part of the statement.

Time Complexity

We can determine the number of steps needed by a program to solve a particular problem instance in 2 ways-

- Step count method
- Tabular method/ table build method

Step Count Method

- We introduce a new variable, *count* , into the program
- *Count* is a global variable, with initial value zero
- Statements to increment *count* by the appropriate amount are introduced into the program.
- Each time a statement in the original program is executed, *count* is incremented by the step count of that statement

Algorithm Sum(a,n)

{

S=0;

Count=count +1; *// for assignment*

for i=1 to n do

{

Count=count +1; *// for*

S=s+a[i];

Count=count+1; *// for assignment*

}

Count=count +1; *// for last time for*

Count=count+1; *// for return*

Return s;

}

One time

n time

n time

One time

One time

Total=2n+3

Algorithm Add(a,b,c,m,n)

{

for i= 1 to m do

{

Count=count +1 ; // for i

m times

for j = 1 to n do

{

Count=count +1 ; // for j

mn times

c[i,j]=a[i,j]+b[i,j];

Count=count +1; // for assignment

mn times

} Count=count +1 ; // for j

m times

Count=count +1; // for i

}

1 times

}

Total = $2mn + 2m + 1$

Algorithm Add(a,b)

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

Diagram illustrating the frequency of execution for each line of the algorithm:

- Line 3: **Count=count +1 ;** (n times)
- Line 5: **Count=count +1 ;** (n times)
- Line 6: **c[i,j]=a[i,j]+b[i,j];** (n times)
- Line 7: **Count=count +1 ;** (n times)
- Line 9: **Count=count +1 ;** (n times)
- Line 11: **Count=count +1 ;** (1 time)

$$\text{Total} = 2n^2 + 2n + 1$$

Algorithm Rsum(a,n)

```
{  
    Count=count +1// for if  
    if(n<=0) then  
    {  
        Count=count +1// for return  
        return 0.0;  
    }  
    else  
    {  
        Count=count +1// for the addition, function invocation  
        and return  
        return Rsum(a,n-1)+a[n];  
    }  
}
```

According to step count is increment by 1, but Rsum affects count. To avoid this problem we introduce another method called step per execution

$$t_{\text{RSum}}(n) = \begin{cases} 2 & \text{if } n=0 \\ 2 + t_{\text{RSum}}(n-1) & \text{if } n>0 \end{cases}$$

Solution

$$\begin{aligned} t_{\text{RSum}}(n) &= 2 + t_{\text{RSum}}(n-1) \\ &= 2 + 2 + t_{\text{RSum}}(n-2) \\ &= 2(2) + t_{\text{RSum}}(n-2) \\ &= 2(2) + 2 + t_{\text{RSum}}(n-3) \\ &= 3(2) + t_{\text{RSum}}(n-3) \\ &\vdots \\ &= n(2) + t_{\text{RSum}}(0) \\ &= 2n + 2 \end{aligned}$$

- When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count, for example,

$$t_{\text{RSum}}(n) = \begin{cases} 2 & \text{if } n=0 \\ 2 + t_{\text{RSum}}(n-1) & \text{if } n>0 \end{cases}$$

- These recursive formulas are referred to as **recurrence relations**. One way of solving any recurrence relation is to make repeated substitutions for each occurrence of the function t_{RSum} on the right-hand side until all such occurrence disappear:

Solution

$$t_{\text{RSum}}(n) = 2 + t_{\text{RSum}}(n-1)$$

$$= 2 + 2 + t_{\text{RSum}}(n-2)$$

$$= 2(2) + t_{\text{RSum}}(n-2)$$

$$= 2(2) + 2 + t_{\text{RSum}}(n-3)$$

$$= 3(2) + t_{\text{RSum}}(n-3)$$

.

.

.

$$= n(2) + t_{\text{RSum}}(0)$$

$$= 2n + 2 \quad n \geq 0$$

So the step count for Rsum is **$2n+2$**

Table build method

- The second method to determine the step count of an algorithm is to build a table in which we list **the total number of steps contributed by each statement**.
- This is obtained by first determining the **number of steps per execution (s/e) of the statement** and the **total number of times(frequency) each statement is executed**.
- **Step per execution (s/e)**
- The **s/e of a statement** is the amount by which the count changes as a result of the execution of that statement .
- The term **frequency** will tell us how many times a statement will be executed.
- By combining these two quantities, the total contribution of each statement is obtained. By adding the contributions of all statements, the step count for the entire algorithm is obtained.

| statement | s/e | frequency | Total steps |
|------------------------------|--------------|-----------|-------------|
| 1. Algorithm Sum(a,n) | | | |
| 2. { | | | |
| 3. S=0; | | | |
| 4. for i=1 to n do | | | |
| 5. { | | | |
| 6. S=s+a[i]; | | | |
| 7. } | | | |
| 8. Return s; | | | |
| 9. } | | | |
| | Total | | |

| statement | s/e | frequency | Total steps |
|--------------------------|-------|-----------|-------------|
| 1. Algorithm Sum(a,n) | 0 | - | 0 |
| 2. { | 0 | - | 0 |
| 3. S=0; | 1 | 1 | 1 |
| 4. for i=1 to n do | 1 | n+1 | n+1 |
| 5. { | 0 | - | 0 |
| 6. S=s+a[i]; | 1 | n | n |
| 7. } | 0 | - | 0 |
| 8. Return s; | 1 | 1 | 1 |
| 9. } | 0 | - | 0 |
| | Total | | 2n+3 |

| statement | s/e | frequency | Total steps |
|------------------------------------|-------|-----------|-------------|
| 1. Algorithm Add(a,b,c,m,n) | | | |
| 2. { | | | |
| 3. for i= 1 to m do | | | |
| 4. { | | | |
| 5. for j = 1 to n do | | | |
| 6. { | | | |
| 7. c[i,j]=a[i,j]+b[i,j]; | | | |
| 8. } | | | |
| 9. } | | | |
| 10. } | | | |
| | Total | | |

| statement | s/e | frequency | Total steps |
|-----------------------------|-------|-----------|-------------|
| 1. Algorithm Add(a,b,c,m,n) | 0 | - | 0 |
| 2. { | 0 | - | 0 |
| 3. for i= 1 to m do | 1 | m+1 | m+1 |
| 4. { | 0 | - | 0 |
| 5. for j = 1 to n do | 1 | m(n+1) | mn+m |
| 6. { | 0 | - | 0 |
| 7. c[i,j]=a[I,j]+b[I,j]; | 1 | mn | mn |
| 8. } | 0 | - | 0 |
| 9. } | 0 | - | 0 |
| 10. } | 0 | - | 0 |
| | Total | | 2mn+2m+1 |

| statement | s/e | Frequency | | Total steps | |
|---|-------|-----------|-----|-------------|-----|
| | | n=0 | n>0 | n=0 | n>0 |
| 1. Algorithm Rsum(a,n) 2. { 3. if(n<=0) then 4. return 0; 5. else 6. return Rsum(a,n-1)+a[n] 7. } | | | | | |
| X=trsum(n-1) | Total | | | | |

| statement | s/e | Frequency | | Total steps | |
|------------------------------------|-------|--------------|-----|-------------|-----|
| | | n=0 | n>0 | n=0 | n>0 |
| 1. Algorithm RSum(a,n) | 0 | - | - | 0 | |
| 2. { | 0 | - | - | 0 | |
| 3. if(n<=0) then | 1 | 1 | 1 | 1 | 1 |
| 4. return 0; | 1 | 1 | 0 | 1 | 0 |
| 5. else | 0 | 0 | 0 | 0 | 0 |
| 6. return Rsum(a,n-1)+a[n] | 1+x | 0 | 1 | 0 | 1+x |
| 7. } | 0 | - | - | 0 | 0 |
| | Total | | | 2 | 2+x |
| | | x=tRSum(n-1) | | | |

Types of Algorithm's Analysis

- Apriory analysis of algorithms

- The Apriory analysis means doing an analysis of an algorithm (i.e., measuring the time and memory) **before running it (programming code) on the system.**
- The analysis is done on the basis of different operations such as **assignment, comparisons, iteration, function call, etc.** which are performed by the programming code.

- Aposterior analysis of algorithms

- In Aposterior analysis of algorithm the analysis of an algorithm (i.e., measuring the time and memory) is **done only after running it (programming code) on the system.**
- This analysis depends highly on the **system's performance** and hence it changes from system to system.
- So efficiency of algorithm will not same for all end users.

Time complexity is measured in the following three different cases:-

Worst case complexity- maximum number of steps taken on any instance of size n

Best case complexity- minimum number of steps taken on any instance of size n

Average case complexity- the average number of steps taken on any instance of size n .

Worst case complexity

- maximum number of steps taken on any instance of size n
- Eg: if the key element is to be searched in the list of n elements and key is available as the last element, then searching will require maximum n searches to find out the key
- Let X_n be the set of all possible inputs of size n and $t(I)$ be the time for processing of the problem instance I of X_n . Then the worst case complexity can be defined as

$$W(n) = \text{Max}\{t(I) \mid I \in X_n\}$$

- ie, worst case complexity for the problem of size n is the maximum of all $t(I)$

Best case complexity

- minimum number of steps taken on any instance of size n
- Eg: if the key element is to be searched in the list of n elements and key is available as the first element, then searching will require minimum, ie 1, searches to find out the key
- Let X_n be the set of all possible inputs of size n and $t(I)$ be the time for processing of the problem instance I of X_n . Then the best case complexity can be defined as

$$B(n) = \text{Min}\{t(I) \mid I \in X_n\}$$

- ie, Best case complexity for the problem of size n is the minimum of all $t(I)$

Average case complexity

- the average number of steps taken on any instance of size n .
- Let X_n be the set of all possible inputs of size n and $t(I)$ be the time for processing of the problem instance I of X_n and $\text{prob}(I)$ be the probability of occurrence of problem instance I . Then the average case complexity can be defined as

$$A(n) = \sum \text{Prob}(I) * t(I) \text{ for all } I \in X_n$$

Algorithm Design Goals

- The three basic design goals that one should strive for in a program are:
 - 1. Try to save Time
 - 2. Try to save Space
 - 3. Try to save Face
- A program that runs faster is a better program, so saving time is an obvious goal. Like wise, a program that saves space over a competing program is considered desirable. We want to “save face” by preventing the program from locking up or generating reams of garbled data.

Classification of Algorithms

- If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

| | |
|---|---|
| 1 | Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant. |
|---|---|

Classification of Algorithms

log n

When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction.

Eg: binary search

n

When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

Eg: maximum element in an array

Classification of Algorithms

| | |
|------------|---|
| $n \log n$ | This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving them independently, and then combining the solutions. When n doubles, the running time more than doubles. |
| n^2 | Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold. Eg: bubble sort |

Classification of Algorithms

n^3

Similarly, an algorithm that process triples of data items (perhaps **in a triple–nested loop**) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight fold.

Eg: matrix multiplication

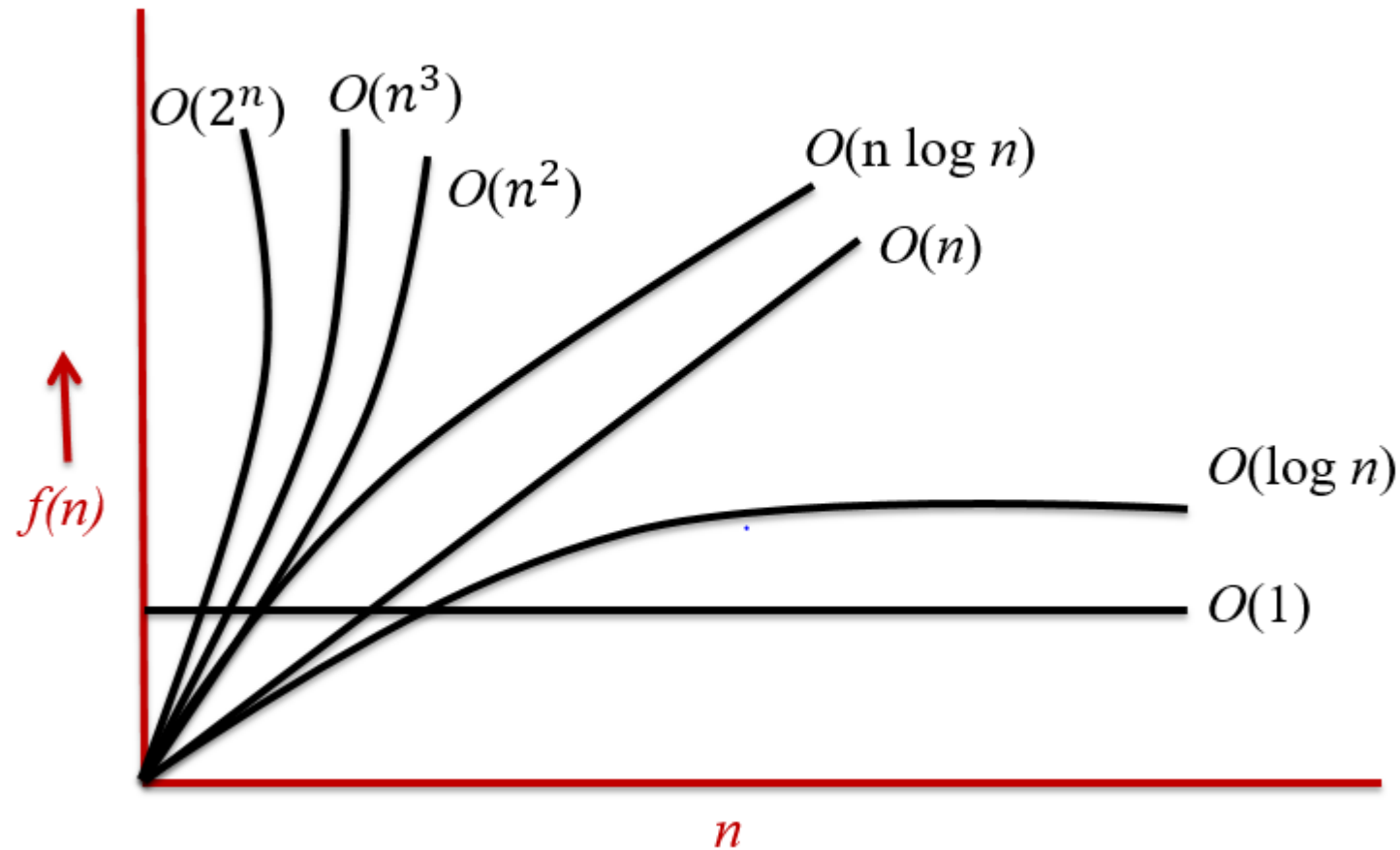
2^n

Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as “brute–force” solutions to problems. Whenever n doubles, the running time squares.

Order of growth

- Time complexity of an algorithm is expressed as a function of the size of the input data of the problem.
- Measuring the efficiency of an algorithm in relation with the input size of the problem is called order of growth
- Order of growth in algorithm means how the time for computation increases when you increase the input size.
- If the running times of two algorithms have the same growth rate then their performance is said to be same but if the running times of two algorithms have different growth rate then their performance significantly changes as the problem input size grows.

Order of growth



From the graph, the logarithmic function is the slowest growing function and the exponential function 2^n grows very fast with varying input size n . The algorithms with comparatively lower degree of functions are called more efficient.

Order of growth

Order of growth of some common functions

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \dots < O(2^n)$$

| n | log₂ n | n*log₂n | n² | n³ | 2ⁿ |
|----------|--------------------------|---------------------------|----------------------|----------------------|----------------------|
| 1 | 0 | 0 | 1 | 1 | 2 |
| 2 | 1 | 2 | 4 | 8 | 4 |
| 4 | 2 | 8 | 16 | 64 | 16 |
| 8 | 3 | 24 | 64 | 512 | 256 |
| 16 | 4 | 64 | 256 | 4096 | 65,536 |
| 32 | 5 | 160 | 1024 | 32,768 | 4,294,967,296 |
| 64 | 6 | 384 | 4096 | 2,62,144 | Note 1 |
| 128 | 7 | 896 | 16,384 | 2,097,152 | Note 2 |
| 256 | 8 | 2048 | 65,536 | 1,677,216 | ???????? |

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(k^n)$ where k is a constant

Asymptotic Notations

- We need some special symbols for comparing the growth rate of functions.
- These special symbols are known as asymptotic notations which are used to describe the running time of an algorithm in terms of functions whose domains are a set of natural numbers
- The asymptotic analysis of algorithm is a means of comparing relative performances of the algorithms.
- **Asymptotic Notations** are notations used for the study of functions of a parameter n , as n becomes larger and larger without bound.

Asymptotic Notation

- **Big Oh** $O()$: *this notation gives an upper bound for a function*
- **Big Omega** or $\Omega()$: *this notation gives a lower bound for a function*
- **Theta** notation or $\Theta()$: *this notation gives tight bound for a function*
- **Little oh** or $o()$: *this notation gives strict upper bound for a function*
- **Little Omega** or $\omega()$: *this notation gives strict lower bound for a function*

Big Oh(O) Notation

- Most commonly used notation to express an algorithm's performance.
- It is a method of expressing the upper bound on the growth rate of an algorithm's running time.
- In other words we can say that it is the longest amount of time an algorithm could possibly take to finish the execution of the algorithm, therefore “Big Oh” or O-notation is used for worst-case analysis of the algorithm.
- Let $f(n)$ describe an algorithm's worst-case performance for some input of size n . Then the Big Oh notation can be defined as:

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

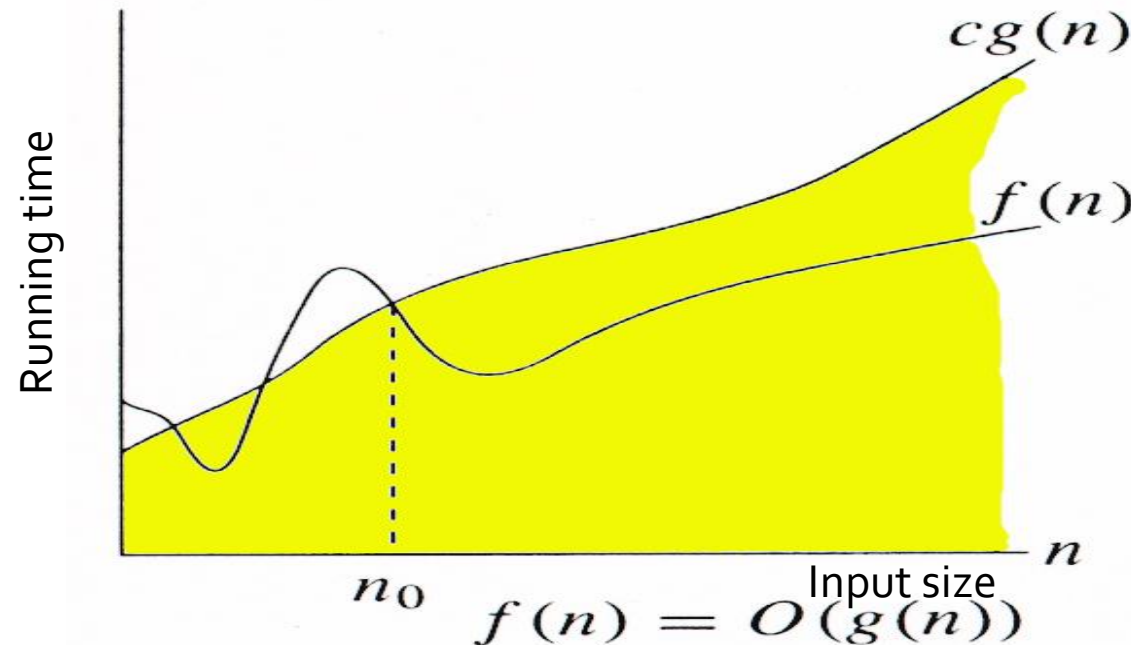
$$f(n) \leq c * g(n) \text{ for all } n, n \geq n_0$$

Mathematically

$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \forall n \geq n_0\}$

- The running time of an algorithm is $O(g(n))$ if whenever the input size is equal to or exceeds some threshold n_0 , its running time can be bounded above by some positive constant 'c' times $g(n)$.

**Graphical
representation
of Big Oh (O)
notation**



The Big Oh notation can also be proved by applying the limit formula

If $\lim_{n \rightarrow \infty} \left(\frac{f(n)}{g(n)} \right)$ exists, then the function $f(n)$ is said to be equal to $O(g(n))$ if

$$\lim_{n \rightarrow \infty} \left(\frac{f(n)}{g(n)} \right) = c \text{ such that } 0 \leq c < \infty$$

Big Oh notation can be summarized as follows:

- The Big Oh notation is an asymptotic upper bound on the growth of an algorithm.
- The Big Oh notation is used widely to characterize algorithm's running times.
- $f(n)$ and $g(n)$ are functions over non-negative integers
- $f(n)=O(g(n))$ is read as “ $f(n)$ is Big Oh of $g(n)$ ”.
- $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically less than or equal to $g(n)$.
- The statement “ $f(n)$ is Big Oh of $g(n)$ ” means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.

Explanation: The two differentiable function $f(n)$ and $g(n)$

$$f(n) \leq c * g(n)$$

$f(n)$ grows with same rate or slower than $g(n)$

$$n \geq n_0, \quad c > 0, \quad n_0 > 1$$

$F(n) = O(g(n))$ logically broken down

$$F(n) = \Theta(g(n)) \quad \text{and} \quad F(n) = o(g(n))$$

Same rate $\{\text{thetta}\}$ slower than $g(n)$ $\{\text{small Oh}\}$

Always the Big Oh notation is slower, $cg(n)$ is upper bound

$g(n)$ is an asymptotic upper bound for $f(n)$

Problem 1:

$$f(n) = 3n + 2$$

$$g(n) = n$$

$$f(n) = O(g(n))$$

$$f(n) \leq c * g(n) \quad c > 0, n_0 > 1 \quad \text{-----}(1)$$

$$3n + 2 \leq c n \quad \text{Assume that } c = 4$$

$$3n + 2 \leq 4n$$

$$n \geq 2$$

Problem 2:

$$f(n) = 2n + 3$$

$$f(n) \leq c * g(n)$$

$$f(n) = O(n)$$

$$2n + 3 \leq 10n \text{ or } 7n \quad (\text{should not write multiple terms})(\text{always greater})$$

$$\text{or } 2n + 3 \leq 2n + 3n$$

$$2n + 3 \leq 5n \quad n \geq 1$$

Or

$$f(n) = 2n + 3$$

$$2n + 3 \leq 2n^2 + 3n^2$$

$$2n + 3 \leq 5n^2 \quad n \geq 1 \quad (\text{also true})$$

$$\text{Here } g(n) = n^2$$

$$F(n) = O(n)$$

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

(both upper bound values)(try to write closest function)

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement “ $f(n)$ is $O(g(n))$ ” means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

| | $f(n)$ is $O(g(n))$ | $g(n)$ is $O(f(n))$ |
|-------------------|---------------------|---------------------|
| $g(n)$ grows more | Yes | No |
| $f(n)$ grows more | No | Yes |
| Same growth | Yes | Yes |

$$F(n) \leq c g(n)$$

Big Omega Ω

- The Big Omega Ω notation is a method of expressing the lower bound on the growth rate of an algorithm's running time.
- In other words, we can say that it is the minimum amount of time, an algorithm could possibly take to finish it.
- Used for best-case analysis of the algorithm.
- If $f(n)$ describes the algorithm's best-case performance for some input of size n , then the Big Omega notation can be defined as
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is omega of $(g(n))$ if there are positive constants c and n_0 such that $\mathbf{f(n) \geq cg(n)}$ for all $n, n \geq n_0$. that is
$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ \mathbf{0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0}$$
- The running time of an algorithm is $\Omega(g(n))$

- If whenever the input size is equal to or exceeds some threshold n_0 , its running time can be bounded above by some positive constant 'c' times $g(n)$.
- An algorithm's performance is measured by the number of primitive operations performed by it, so Big Omega notation provides information on the minimum number of operations that an algorithm needs to perform in order to solve the problem
- So if the complexity of any algorithm is $\Omega(n)$ it means the algorithm has to do at least cn operations.

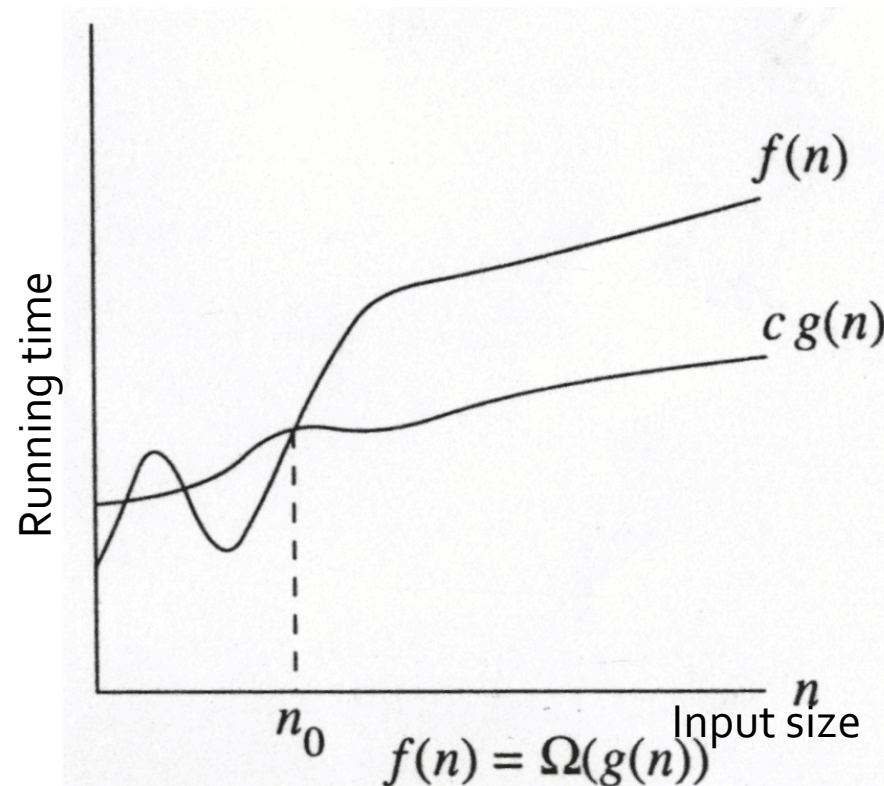
- Lower bound implies that below this time the algorithm can not perform better
- the inverse of $O(n)$

Graphical representation of Big Omega Ω notation

The Big Omega Ω running time can also be proved by applying the limit formula as

If $\lim_{n \rightarrow \infty} \left(\frac{f(n)}{g(n)} \right)$ exists, then the function $f(n)$ is said to be equal to $\Omega(g(n))$ if

$$\lim_{n \rightarrow \infty} \left(\frac{f(n)}{g(n)} \right) = c \text{ such that } 0 \leq c < \infty$$



Big Omega Ω notation can be summarized as follows:

- The Big Omega notation is an asymptotic lower bound on the growth of an algorithm.
- $f(n)$ and $g(n)$ are functions over non-negative integers
- $f(n) = \Omega(g(n))$ is read as “ $f(n)$ is Big Omega of $g(n)$ ”.
- $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically more than or equal to $g(n)$.
- The statement “ $f(n)$ is Big Omega of $g(n)$ ” means that the growth rate of $f(n)$ is no less than the growth rate of $g(n)$.

Big omega

2 differentiable function $f(n)$ & $g(n)$
 $f(n)$ grows with same rate or faster
than $g(n)$

$$F(n) \geq c * g(n) \quad c > 0, n_0 \geq 1, n \geq n_0$$

$$F(n) = \Omega(g(n))$$

$g(n)$ is an asymptotic lower bound
of $f(n)$

$$F(n) = 3n + 2 \quad g(n) = n$$

$$3n + 2 \geq c * n \quad (\text{Assume } c = 1)$$

$$3n + 2 \geq n \quad n_0 \geq 1$$

$3n + 2 = \Omega(n)$ // formula for calculating
time complexity of big Omega
notation

$$F(n) = \Omega(g(n))$$

$$O(1) < O(\log n) < O(n) < o(n \log n) < o(n^2) < o(n^3) < \dots < o(2^n)$$

lower

upper

$$F(n) = 2n + 3 \quad g(n) = n$$

$$F(n) = \Omega(g(n))$$

$$F(n) \geq c * g(n) \quad (\text{Assume } c = 1)$$

$$2n + 3 \geq 1 * n$$

$$2n + 3 \geq n$$

$$F(n) = \Omega(n)$$

$$2n + 3 \geq 1 * \log n \quad (\text{also true})$$

$$2n + 3 \geq \log n$$

$$F(n) = \Omega(\log n)$$

$$F(n) = \Omega(n^2) \text{-----false}$$

Theta Notation Θ

- The Theta Notation Θ is a method of expressing the asymptotic **tight bound** on the growth rate of an algorithm's running time both from above and below, that is upper bound and lower bounds.
- In other words we can say that it describes the minimum as well as maximum amount of time an algorithm could possibly take to finish it.

There fore,

If $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$, then $f(n)=\Theta(g(n))$

Or

$f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$ then $f(n), f(n)=\Theta(g(n))$

- The “Theta” or Θ notation is used for average case analysis of an algorithm.

- If $f(n)$ describes the algorithm's average-case performance for some input of size n , then the theta notation is defined as:

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is Theta of ($g(n)$) if there are positive constants c_1, c_2 and n_0 such that

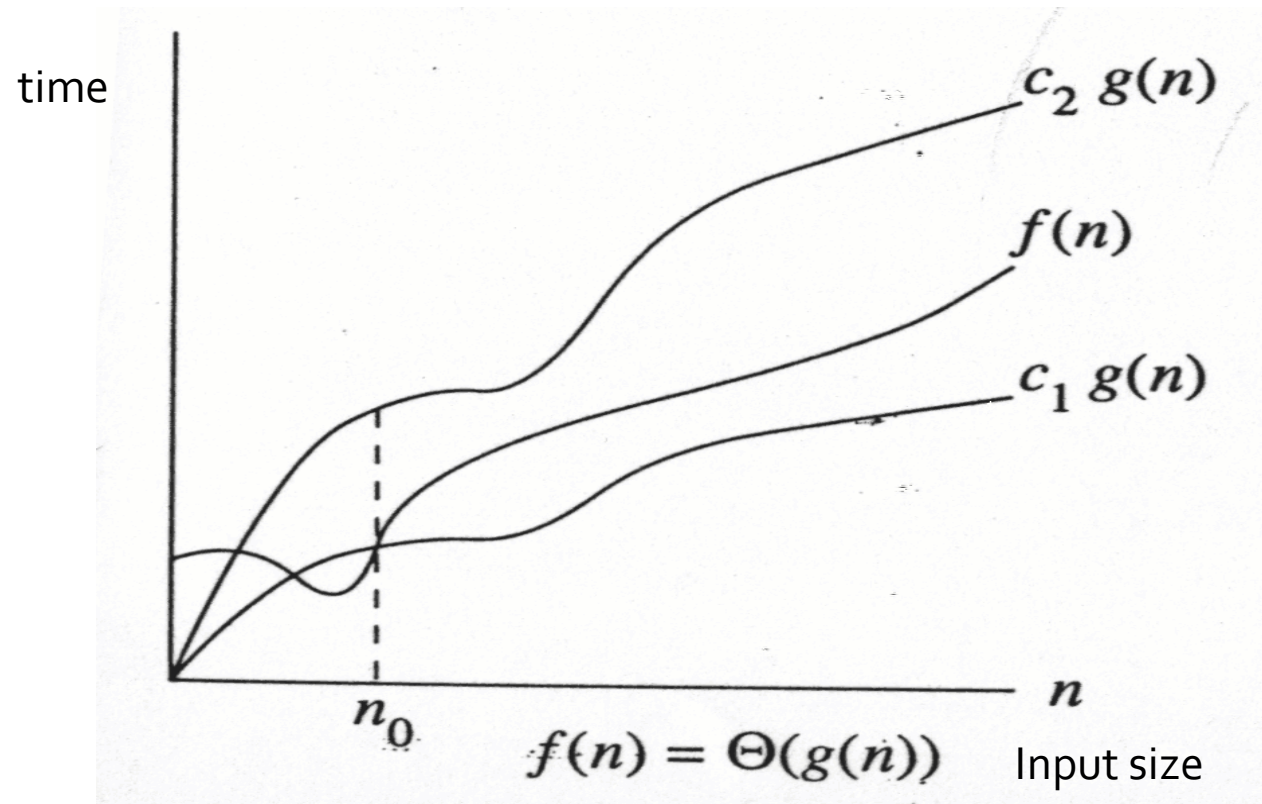
$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n, n \geq n_0.$$

- $f(n)$ lies between c_1 times the function g and c_2 times the same function g except possibly when n is smaller than n_0 .
- $\Theta(g(n)) = \{f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that}$
$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0$$
- The theta notation is more precise than both the Big Oh and Big Omega notations. The function $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.

The Theta (Θ) running time can also be proved by applying the limit formula as

If $\lim_{n \rightarrow \infty} \left(\frac{f(n)}{g(n)} \right)$ exists, then the function $f(n)$ is $\Theta(g(n))$ if $\lim_{n \rightarrow \infty} \left(\frac{f(n)}{g(n)} \right) = c$ such that $0 < c < \infty$

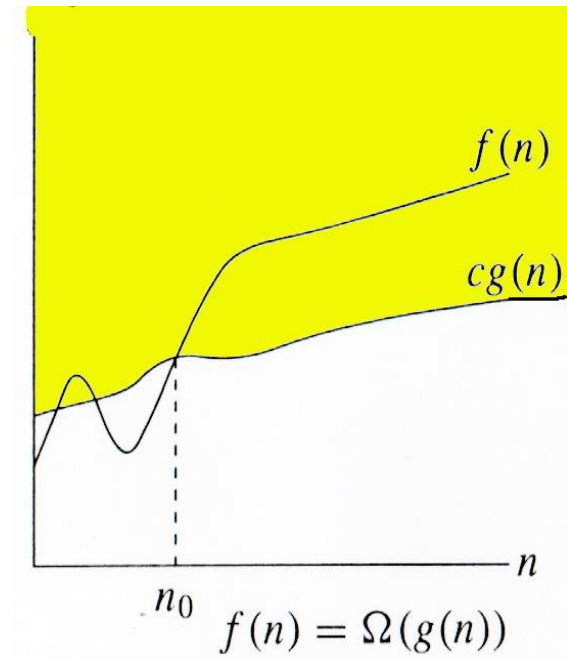
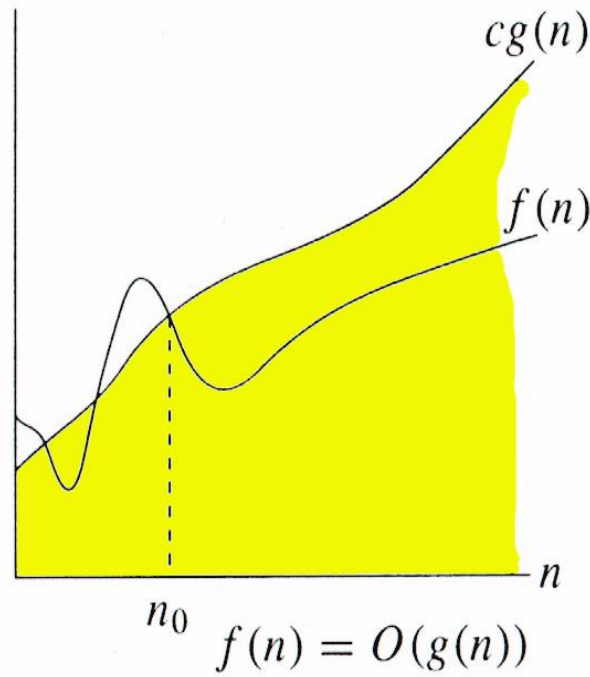
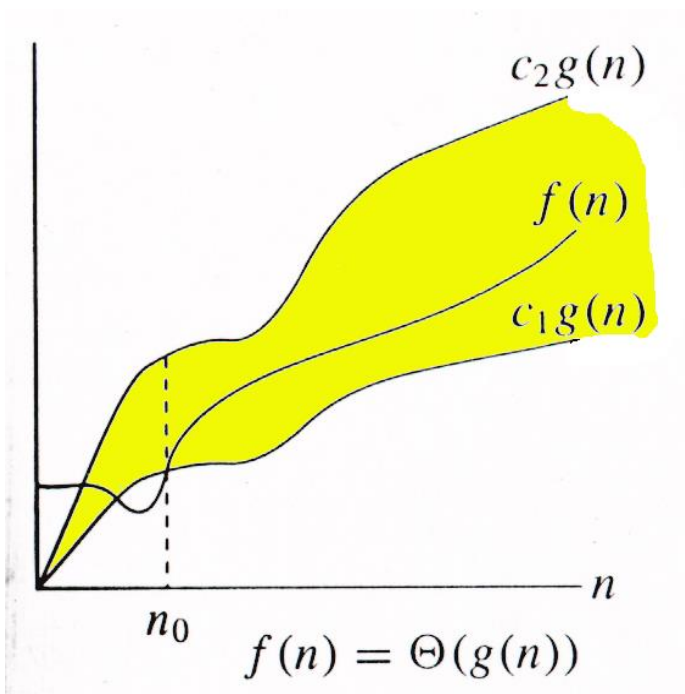
Graphical representation of Theta (Θ) notation



Relation Between Θ , O , Ω

1. The Big Oh(O) is used to measure the upper bound on the algorithm's growth. The Big Oh(O) defines the “order at most” which means $c \times g(n)$ is an upper bound on $f(n)$ when $f(n)=O(g(n))$
2. The Big Omega (Ω) is used to measure the lower bound on the algorithm's growth. The Big Omega(Ω) defines the “order at least” which means $c \times g(n)$ is a lower bound on $f(n)$ when $f(n)=\Omega(g(n))$
3. The Theta(Θ) is used to measure the lower as well as the upper bound on the algorithm's growth. The Theta(Θ) defines the “order exactly” which means $c_1 \times g(n)$ is an upper bound on $f(n)$ and $c_2 \times g(n)$ is a lower bound on $f(n)$ when $f(n)=\Theta(g(n))$.

Graphical examples of Θ , O , Ω



Theorem : For any two functions $g(n)$ and $f(n)$,
 $f(n) = \Theta(g(n))$ iff
 $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

i.e., $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

In practice, asymptotically tight bounds are obtained from asymptotic upper and lower bounds.

Theta Notation

The function $f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_0

Such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$

Example: $f(n) = 2n + 3$

$$\begin{array}{ccccccc} 1 & \times & n & \leq & 2n + 3 & \leq & 5 \times n & & f(n) = \Theta(n) \\ | & & | & & | & & | & & \\ | & & | & & | & & | & & \\ c_1 & g(n) & f(n) & & & & c_2 & g(n) \end{array}$$

Which means $2n + 3$ is exactly $O(n)$ order

We choose $f(n)$ and $g(n)$ as two differential functions and say that they have same growth rate

Theta Notation

$$f(n)=3n+2 \quad g(n)=n$$

$c_1g(n) \leq f(n) \leq c_2g(n)$ divide into two parts

$$f(n) \leq c_2(g(n)) \qquad f(n) \geq c_1(g(n))$$

$$3n+2 \leq c_2 n \qquad 3n+2 \geq c_1 n$$

$$3n+2 \leq 4n \text{ (upper bound, } c_2=4) \qquad 3n+2 \geq 1n \text{ (lower bound, } c_1=1)$$

$$n_0 \geq 1$$

Little Oh (o)

- It is the method of expressing the upper bound on the growth rate of an algorithm's running time which may or may not be asymptotically tight.
- It is called a loose upper bound.
- We use Little Oh(o) notations to denote upper bound that is asymptotically not tight.
- Little Oh(o) can defined as:
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is Little Oh of ($g(n)$). If there are positive constants c and n_0 such that $\mathbf{f(n)} < \mathbf{cg(n)}$ for all $n, n \geq n_0$. that is f has a lower growth rate than g .
- So Little Oh is to mean “Tight upper bound”
- Difference between Big Oh and Little Oh???

- In Big Oh $f(n)=O(g(n))$ and the bound is $0 \leq f(n) \leq cg(n)$ so it is true only for some positive value of c .
- But in case of Little Oh, it is true for all constant $c>0$ because $f(n)=o(g(n))$ and the bound $f(n)<cg(n)$.
- Little Oh(o) can also defined as

$$g \in o(f) \text{ if } \lim_{n \rightarrow \infty} \left(\frac{f(n)}{g(n)} \right) = 0$$

Little Omega (ω)

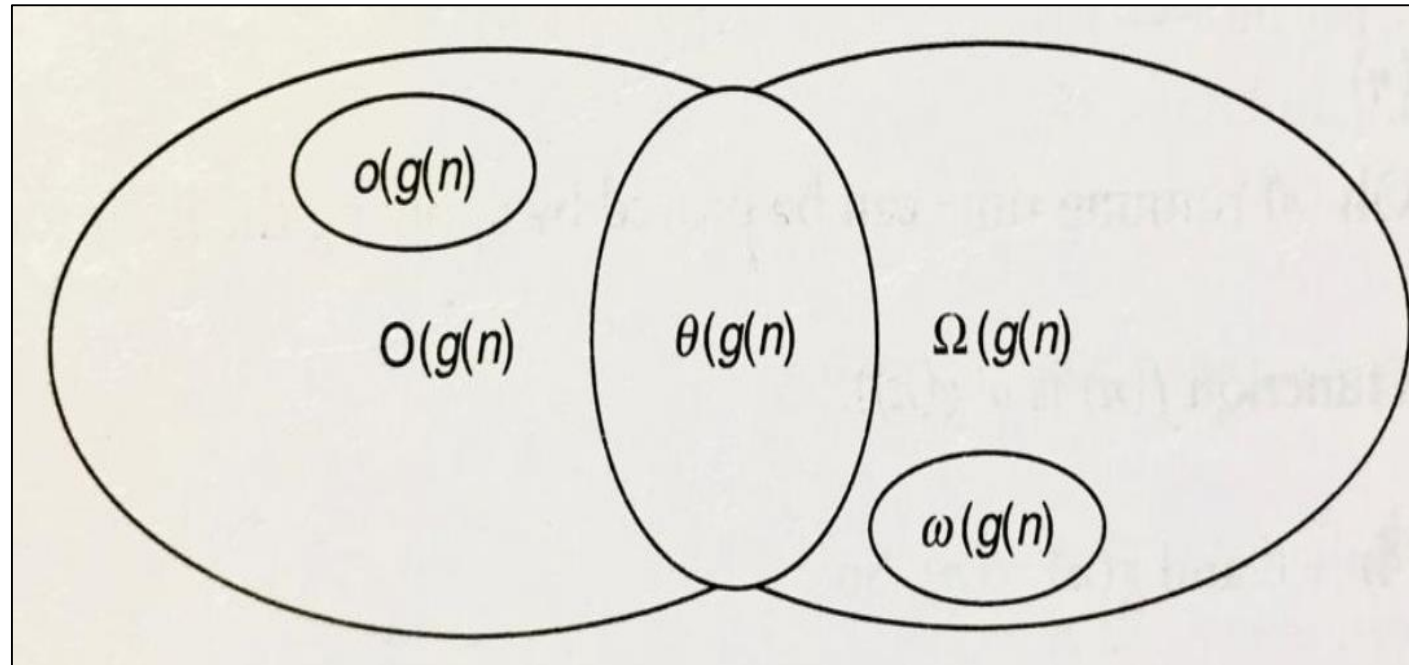
- It is the method of expressing the lower bound on the growth rate of an algorithm's running time which may or may not be asymptotically tight.
- It is called a loose lower bound.
- We use Little Omega (ω) notation to denote lower bound that is asymptotically not tight.
- Little Omega (ω) can be defined as:
 - Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is Little Omega of ($g(n)$).
If there are positive constants c and n_0 such that $\mathbf{f(n)} > \mathbf{cg(n)}$ for all $n, n \geq n_0$.
 - That is f has a higher growth rate than g .
- So Little Omega (ω) is to mean “Tight lower bound”
- Difference between Big Omega and Little Omega???

- In Big Omega $f(n)=\Omega(g(n))$ and the bound is $0 \leq cg(n) \leq f(n)$ so it is true only for some positive value of $c>0$.
- But in case of Little Omega (ω), it is true for all constant $c>0$ because $f(n)=o(g(n))$ and the bound $f(n)>cg(n)$.
- Little Omega (ω) can also defined as

$$g \in \omega(f) \text{ if } \lim_{n \rightarrow \infty} \left(\frac{f(n)}{g(n)} \right) = \infty \quad // \quad \lim_{n \rightarrow \infty} \left(\frac{g(n)}{f(n)} \right) = 0$$

Relation between Big Oh(O), Big Omega(Ω), Theta(Θ), Little Oh(o) and Little Omega(ω)

1. The Big Oh(O) notation is used to measure the upper bound on the algorithm's growth.
2. The Big Omega(Ω) notation is used to measure the lower bound on the algorithm's growth.
3. The Theta notation(Θ) is used to measure the lower as well as upper bound on the algorithm's growth.
4. The Little Oh(o) notation is used to denote upper bound that is asymptotically not tight.
5. The Little Omega(ω) notation is used to denote lower bound that is asymptotically not tight.



Venn diagram showing the relationship between Big Oh(O), Big Omega(Ω), Theta(Θ), Little Oh(o) and Little Omega(ω)

From the Venn diagram,

1. $o(g(n))$ is a subset of $O(g(n))$.
2. $\omega(g(n))$ is a subset of $\Omega(g(n))$.
3. $\Theta(g(n))$ is the intersection of $O(g(n))$ and $\Omega(g(n))$.
4. The union of $\omega(g(n))$ and $\Theta(g(n))$ is a subset of $\Omega(g(n))$.
5. The union of $o(g(n))$ and $\Theta(g(n))$ is a subset of $O(g(n))$.

Recursion

- Recursion is an ability of an algorithm to repeatedly call itself until a certain condition is met. Such condition is called the **base condition**. The algorithm which calls itself is called a recursive algorithm.
- A recursive algorithm is said to be well-defined if the base condition is defined. Each time when the algorithm calls itself it must be closed to the base condition, otherwise recursion will not terminate and would run indefinitely.
- The recursive algorithms must satisfy the following two conditions:
 1. It must have the base case: The value of which algorithm does not call itself and can be evaluated without recursion.
 2. Each recursive call must be to a case that eventually leads toward a base case.

- Factorial of 5

$$5! = 5 * 4!$$

$$4 * 3!$$

$$3 * 2!$$

$$2 * 1!$$

$$1 * 0!$$

$$1$$

- Recursion requires more memory and time to implement. Therefore, the program implemented without recursion may work faster. So the factorial of a given number can be obtained either by using iterative method or recursive method.

Recurrences Relation

- An algorithm is said to be recursive if it can be defined in terms of itself. The running time of recursive algorithm is expressed by means of recurrence relations.
- A recurrence relation is an equation of inequality that describes a function in terms of its value on smaller inputs. It is a recursive function of the size of the problem.
- It is generally denoted by $T(n)$ where n is the size of the input data of the problem.
- The recurrence relation satisfies both the conditions of recursion, that is, it has both the base case as well as the recursive case.
- Therefore, there should be some base value for $T(n)$ and it should be defined in terms of its earlier values in the sequence like $T(n-1)$.
- The portion of the recurrence relation that does not contain T is called the **base case of the recurrence relation** and the portion of the recurrence relation that contains T is called the **recursive case of the recurrence relation**

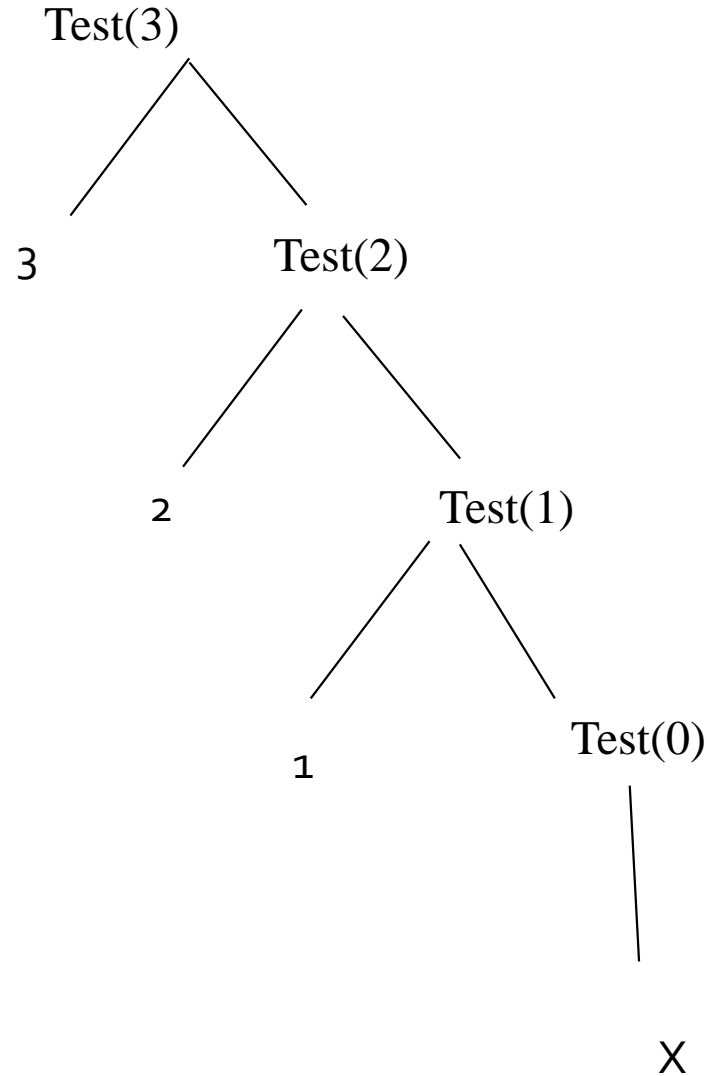
Solving Recurrence Relations

There are four methods for solving Recurrence Relation:

- Substitution Method
- Iteration Method
- Recursion Tree Method
- Master Method

```
Void Test(n)
{
    if(n>0)
    {
        print(n);
        Test(n-1);
    }
}
```

Suppose I am calling this function with $n=3$



Tracing tree for this function/Recursive tree

3+1 times

If we use n function call $n+1$ times

Time complexity= $O(n)$ - order

We can use $O(n)/\Omega(n)/Q(n)$

```

Void Test(n)  _____ T(n)
{
  if(n>0) _____ Ignore/constant =1
  {
    print(n); _____ 1 unit
    Test(n-1); _____ T(n-1)
  }
}

```

Total time , $T(n)=T(n-1)+1$

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$$

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

Substitute $T(n-1)$

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

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

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

$$T(n) = T(n-3) + 3$$

.

. K times

.

.

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

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

$$T(n-2) = T(n-3) + 1$$

$$T(n) = T(n-k) + k$$

We have to reach $n=0$ ie, value=1

Assume that $n-k=0$; ie, $n=k$

Substitute value of $k=n$

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

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

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

$T(n) = n+1$ we get $n+1$ calls

Order (n), linear condition/class

Theta notation ie, $\Theta(n)$

Substitution Method

- Based on the guessing of the solution and then using mathematical induction, it is proved that the guessed solution is correct.
- The guessed solution is substituted for the function when the inductive hypothesis is applied to smaller values. So it is named as substitution method.
- This method is used to establish either upper or lower bounds on a recurrence relation.
- Powerful method, but it can be only be applied when it is easy to guess the solution.

Steps for solving recurrence using substitution method:

1. Guess the solution: It requires making the intelligent guesswork about the solution of a recurrence relation.

a) The table of some of the function values can be constructed and the equation can be formed based on the entries of the table.

b) The good guess of the solution can be made by considering the given recurrence relation.

$T(n)=aT(n/b)+f(n)$ under three different cases:

Case 1: if $a=2$ and $b=2$, then $T(n)=O(f(n) \log(n))$ can be the appropriate guess.

Case 2: if $a=1$ and $b=\text{any arbitrary value}$, then $T(n)=O(\log(n))$ can be the appropriate guess.

Case 3: if $a \neq 2$ and $b > a$, then $T(n)=O(f(n) \cdot (n))$ can be the appropriate guess.

Note: the constant term added to the right-hand side of the recurrence do not affect the solution to the recurrence. Therefore , the guessing of the solution remains the same.

c) The recursion tree methods also be the appropriate guess.

d) If the recurrence is similar to the recurrence you have already come across, then guessing a similar solution is reasonable.

2. By using the mathematical induction, find the constant value of c and n_0 by substituting the guess and prove that our guess is the correct solution.

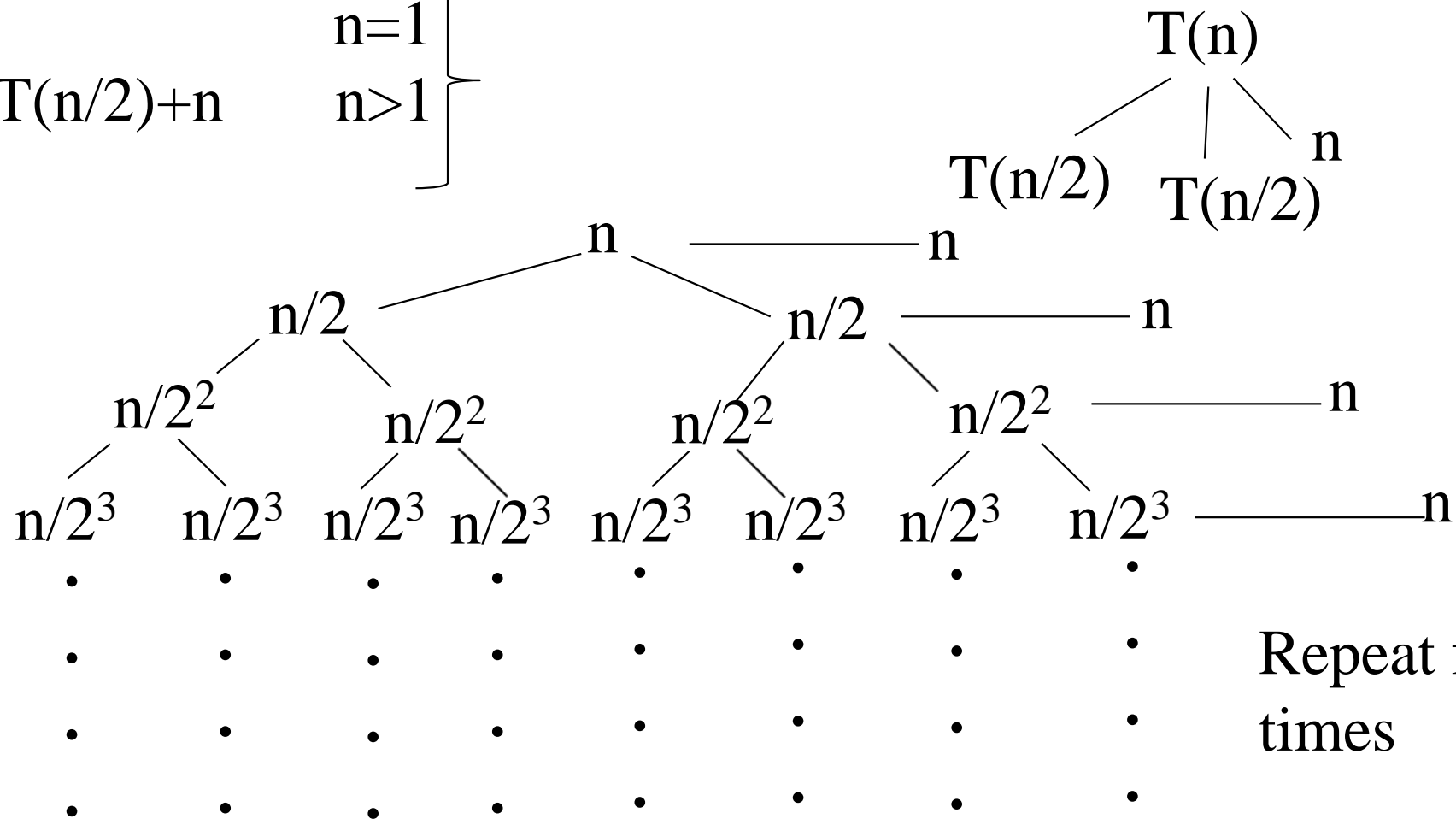
Solve the recurrence $T(n)=2T(n/2)+n$ using the substitution method

void Test(int n) ----- $T(n)$

```
{  
    if(n>1) {  
        for(i=0;i<n;i++)  
        {  
            stmts;-----n  
        }  
        Test(n/2);----- $T(n/2)$   
        Test(n/2);----- $T(n/2)$   
    }  
}
```

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

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



Assume that $n/2^k = 1$

$n = 2^k$, $k = \log n$

ie, nk

$n \log n$

$O(n \log n)$

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

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

$$= 2[2T(n/2^2) + n/2] + n$$

$$= 2^2 T(n/2^2) + n + n \text{ -----(2)}$$

$$= 2^2 [2T(n/2^3) + n/2^2] + 2n$$

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

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

This will continue for k times

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

Assume

$$T(n/2^k) = T(1)$$

$$\text{ie, } n/2^k = 1 \quad n = 2^k$$

$$k = \log n$$

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

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

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

$$T(n) = 2^k T(1) + kn$$

$T(n) = n \times 1 + n \log n$ (we take $n \log n$ (highest of these))

$T(n) = O(n \log n)$ we can use Big Oh or theta notation

Iteration Method

- In the iterative method for solving recurrence relation, the recurrence relation is expanded using backward substitution and then some standard general algebraic formula is applied to express the expanded recurrence relation in the form of summation of terms dependent only on 'n' and some initial conditions.
- The recurrence relation is iterated until the initial condition is reached.
- This is also known as bounding the recurrence relation using known mathematical series.

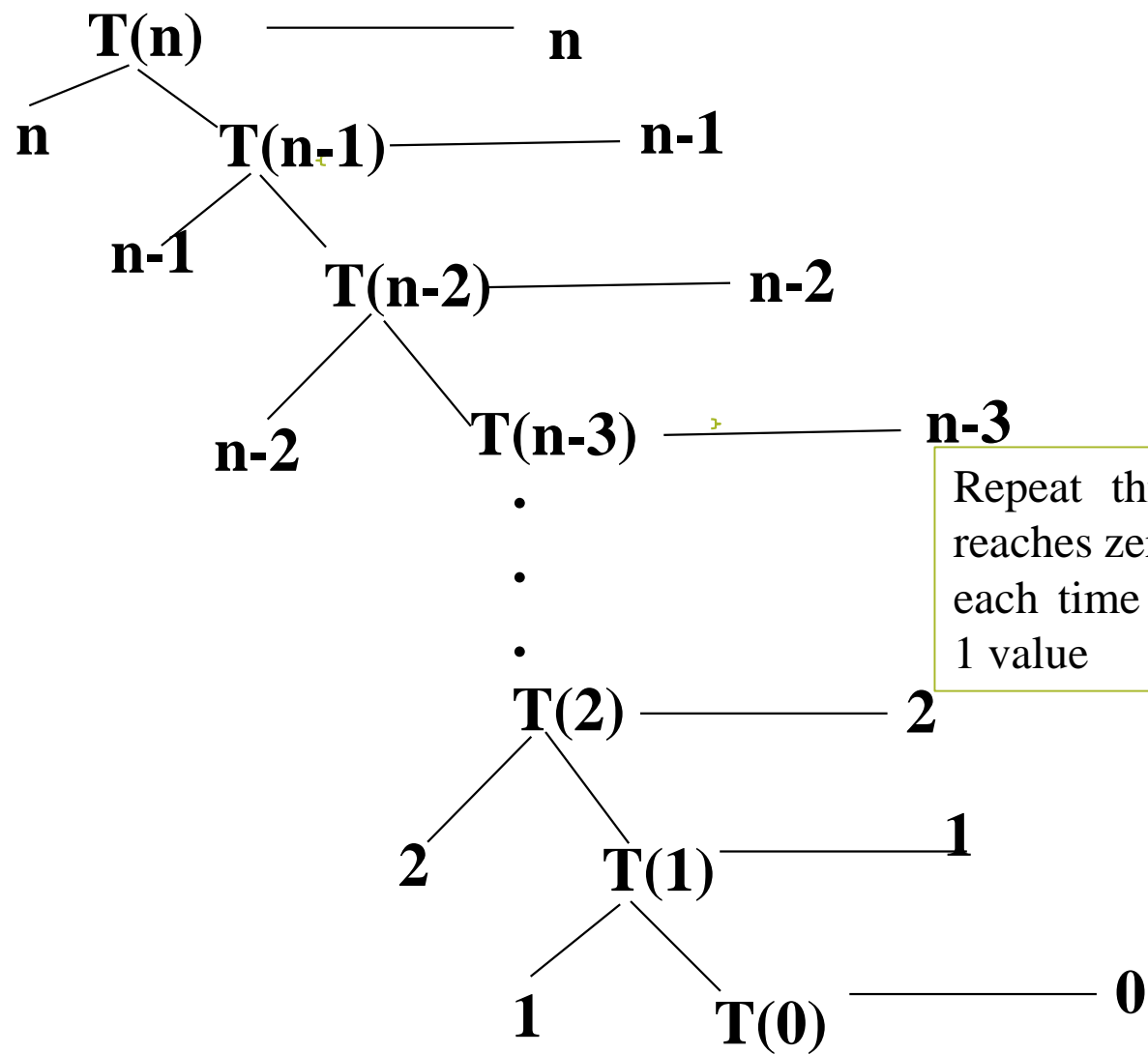
Solve the Recurrence Relation $T(n)=T(n-1)+n$ using Iterative Method

```
Void Test(int n)  _____  T(n)
{
    if(n>0) _____ 1 unit of time
    {
        for(i=0;i<n;i++) _____ n+1
        {
            printf("%d",n); _____ n
        }
        Test(n-1); _____ T(n-1)
    }
}
```

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

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

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



$$T(n) = \begin{cases} 1 & n=1 \\ T(n-1)+n & n>0 \end{cases}$$

Repeat this until it reaches zero.
each time it reduces 1 value

$$0+1+2+\dots+n-3+n-2+n-1+ = \frac{n(n+1)}{2}$$

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

$$O(n^2)$$

$$T(n) = \begin{cases} 1 & n=1 \\ T(n-1)+n & n>0 \end{cases}$$

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

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

$$T(n) = T(n-2) + \underbrace{(n-1) + n}_{\text{red bracket}} \text{ -----(2)}$$

! Remember: Don't add these. we need a sequence to solve so better avoid adding these.

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

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

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

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

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

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

In general

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

! Assume that we reach $n-k=0$

Assume $n-k=0$

$$n=k$$

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

$T(n)=T(0)+1+2+3+\dots+n-1+n$ //sum of first n natural numbers

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

ie, $O(n^2)$

Recursion Tree Method

- In this method, we draw a recurrence tree and calculate the time taken by every level of tree.
- Finally, we sum the work done at all levels.
- To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels.
- The pattern is typically a arithmetic or geometric series.

- In recursion tree, the recurrence is converted into a tree where each node represents the cost incurred at various levels of recursion, that is the cost of the sub problems in the set of recursive function invocations.
- Sum of the cost of various nodes at each level is calculated, called as pre-level cost, and then the sum of pre-level costs is obtained to determine the total cost of all the levels of the recursion tree.
- The construction of recursion is started generally from the second term of the recurrence as a root node and the number of children is decided by the coefficient of the first term.
- So if the recurrence relation is $T(n)=bT(n/c)+f(n)$, then $f(n)$ is the root of the tree and each node should have 'b' children.
- Each node of the tree is expanded until we reach the initial condition.

Number of sub problems

Cost incurred for dividing and combining

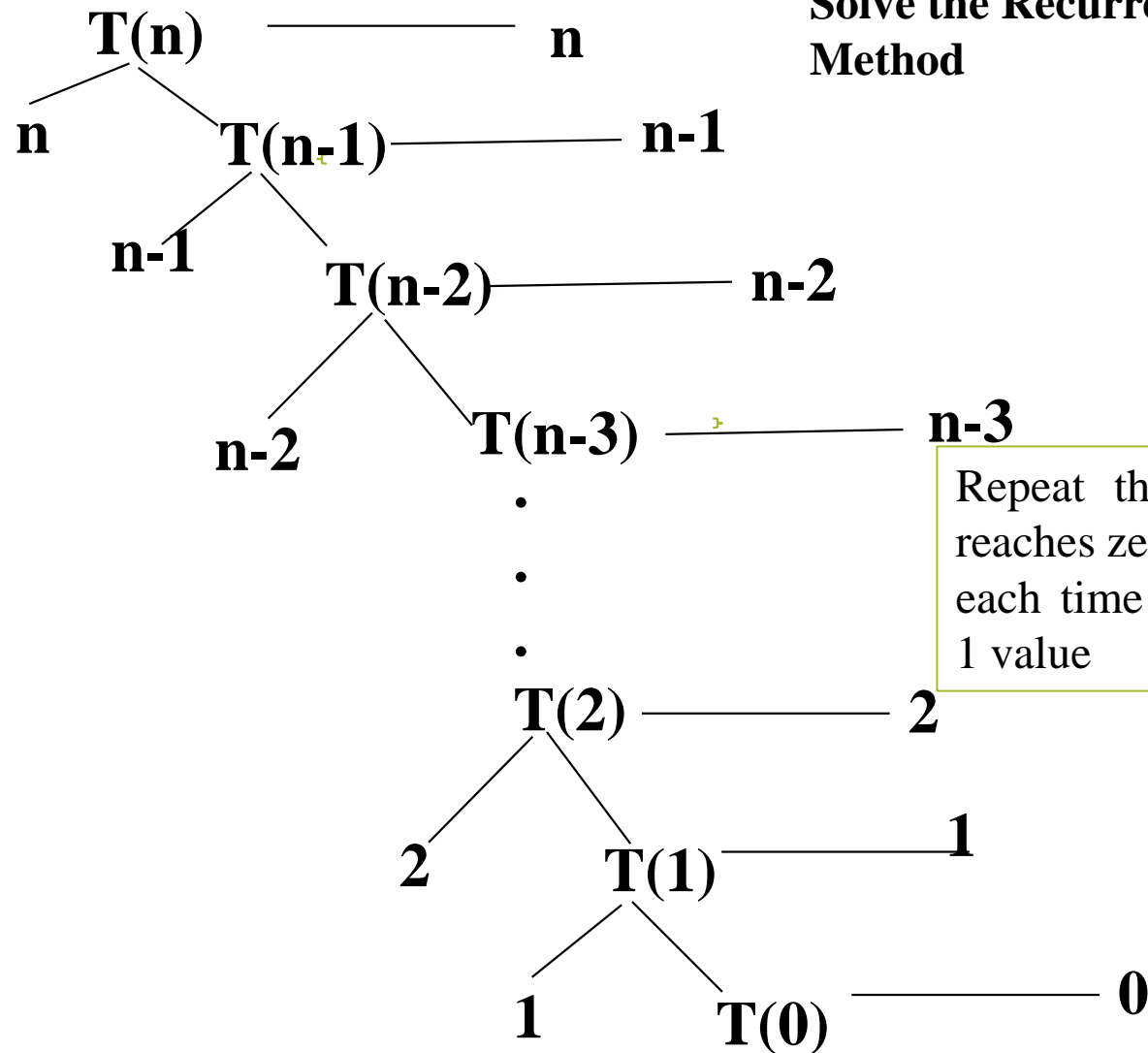
$$T(n) = aT(n/b) + (cn)$$

Size of the sub problem

Figure: Description of the recurrence for recursion tree

- The recursion tree is used when the recurrence relations is defined for the divide and conquer technique ie, it describes the running time of the divide and conquer algorithm.
- The recursion tree is pictorial representation of iterative method and is used to generate a good guess which can be verified by substitution method

Solve the Recurrence Relation $T(n)=T(n-1)+n$ using Recursion Tree Method



$$T(n) = \begin{cases} 1 & n=1 \\ T(n-1)+n & n>0 \end{cases}$$

Repeat this until it reaches zero.
each time it reduces 1 value

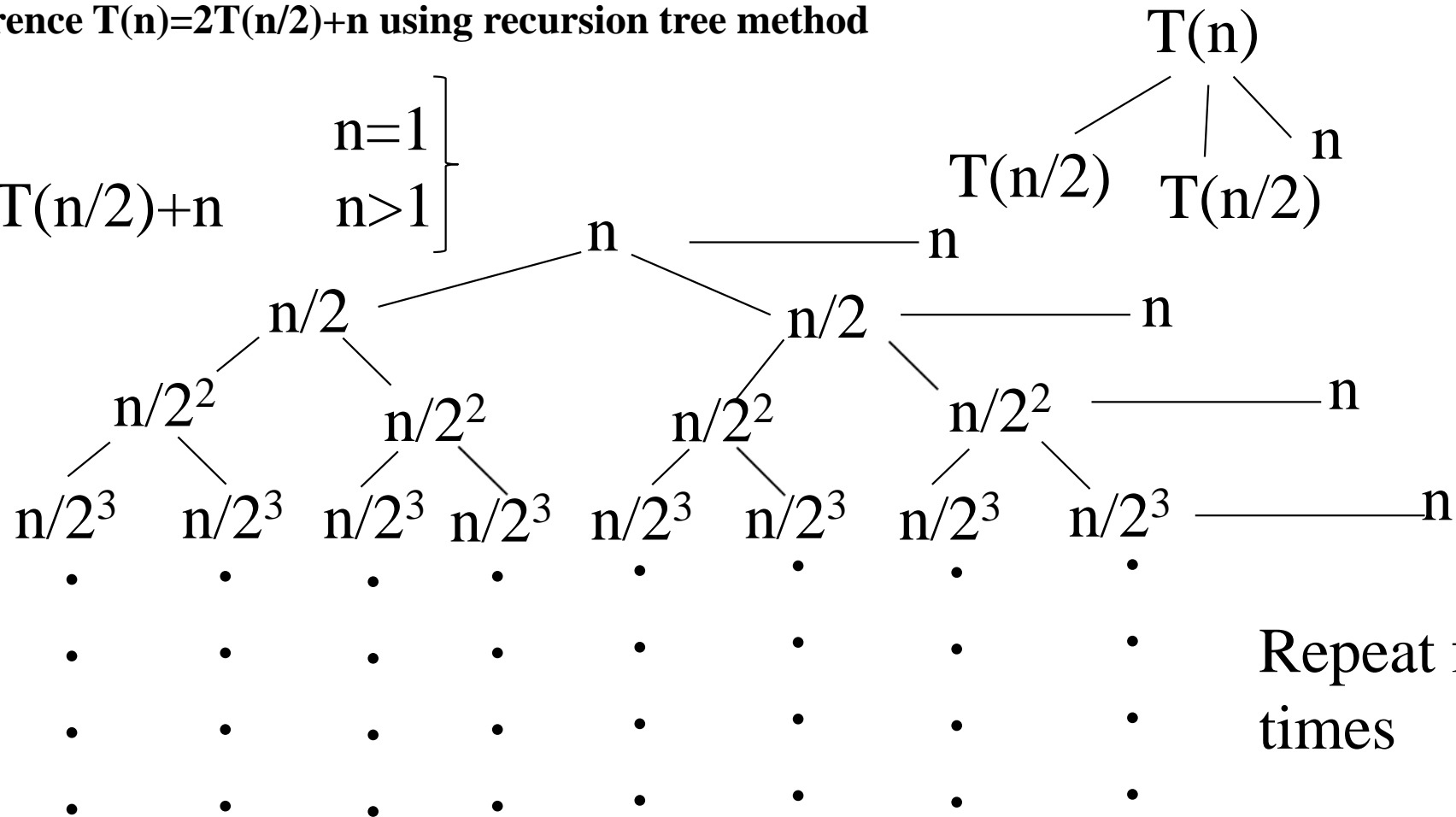
$$0+1+2+\dots+n-3+n-2+n-1+ = \frac{n(n+1)}{2}$$

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

$$O(n^2)$$

Solve the recurrence $T(n)=2T(n/2)+n$ using recursion tree method

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



Assume that $n/2^k=1$

$n=2^k$, $k=\log n$

ie, nk

$n \log n$

$O(n \log n)$

Master's theorem

- Let the recurrence relation be in the form

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

Such that $a \geq 1$ and $b > 1$ be the constants, $f(n)$ be an asymptotically positive function, $T(n)$ be defined on non-negative integers 'n' and $T(n)$ is the run time for an algorithm.

- This is the recurrence relation corresponding to the divide and conquer algorithm such that problem of size n is divided into 'a' sub problems each of size (n/b) . Here ceiling $\lceil n/b \rceil$ and floor $\lfloor n/b \rfloor$ operation can be omitted.
- The recurrence relation $T(n)=aT(n/b)+f(n)$ may be explained as “sub problems of size n/b are solved recursively, each in time $T(n/b)$ where $f(n)$ is the cost of dividing the problem and combining the results”.

$T(n)$ is said to be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b(a)-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b(a)})$. This means that $f(n)$ grows polynomially slower than $n^{\log_b(a)}$ by an n^ϵ so the running time is dominated by cost at leaves.
2. If $f(n) = \Theta(n^{\log_b(a)})$, $k \geq 0$ then $T(n) = \Theta(n^{\log_b(a)} \log^{k+1}(n))$. This means that $f(n)$ and $(n^{\log_b(a)})$ grows at similar rates, so the running time is evenly distributed throughout the tree.
3. If $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some $\epsilon > 0$ and if regularity condition $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and for all $n > n_0$ for some $n_0 > 0$, then $T(n) = \Theta(f(n))$. This means that $f(n)$ grows polynomially faster than $(n^{\log_b(a)})$ by an n^ϵ factor, so running time is dominated by cost at root. Hence this is the inverse of Case 1.

The tree different cases can be summarized as follows

$$T(n) = \left\{ \begin{array}{ll} \Theta(n^{\log_b a}) & \text{when } f(n) = O(n^{\log_b a - \epsilon}) \\ \Theta(n^{\log_b a} \log n) & \text{when } f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & \text{when } f(n) = \Omega(n^{\log_b (a) + \epsilon}) \\ & \text{and } af(n/b) < cf(n) \text{ for large } n \end{array} \right\} \begin{array}{l} \epsilon > 0 \\ c < 1 \end{array}$$

- **A utility method for analysing recurrence relations**
- **Useful in many cases for divide and conquer algorithms**
- **These recurrence relations are of the form:**

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

with $a \geq 1$
and $b > 1$

- **n = the size of the current problem**
- **a = the number of subproblems in the recursion**
- **n/b = the size of each subproblem**
- **$f(n)$ = the cost of the work that has to be done outside the recursive calls (cost of dividing + merging)**

There are 3 cases:

⁺**1. The running time is dominated by the cost at the leaves:**

If $f(n) = O(n^{\log_b(a) - \epsilon})$, then $T(n) = \Theta(n^{\log_b(a)})$
for an $\epsilon > 0$

2. The running time is evenly distributed throughout the tree:

If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n) = \Theta(n^{\log_b(a)} \log(n))$

3. The running time is dominated by the cost at the root:

If $f(n) = \Omega(n^{\log_b(a) + \epsilon})$, then $T(n) = \Theta(f(n))$
for an $\epsilon > 0$

If $f(n)$ satisfies the regularity condition:

$af(n/b) \leq cf(n)$ where $c < 1$ (this always holds for polynomials)

Because of this condition, the Master Method cannot solve every recurrence of the given form.

How to apply the Master Method (step-by-step)

+

$$T(n) = aT(n/b) + f(n)$$
The diagram shows three blue arrows pointing downwards from the equation above. The first arrow points from the coefficient 'a' to the word 'Extract' in step 1. The second arrow points from the term 'n/b' to the variable 'b' in step 1. The third arrow points from the term 'f(n)' to the function 'f(n)' in step 1.

1. Extract a , b and $f(n)$ from a given recurrence.

2. Determine $n^{\log_b(a)}$.


3. Compare $f(n)$ and $n^{\log_b(a)}$ asymptotically.

4. Determine the appropriate Master Method case and apply it.

Imagine that: $T(n) = 9T(n/3) + n$.

1. Extract; $a = 9$, $b = 3$ and $f(n) = n$.

2. Determine; $n^{\log_b(a)} = n^{\log_3(9)} = n^2$.

3. Compare; $n^{\log_b(a)} = n^2$  $f(n) = n$

4. Thus case 1; (Express $f(n)$ in terms of $n^{\log_b(a)}$)

Because $f(n) = O(n^{2 \cdot \epsilon})$,

$T(n) = \Theta(n^{\log_b(a)}) = \Theta(n^2)$.

Imagine that: $T(n) = 2T(n/2) + n$.

1. Extract; $a = 2$, $b = 2$ and $f(n) = n$.

2. Determine; $n^{\log_b(a)} = n^{\log_2(2)} = n^1 = n$.

**3. Compare; $n^{\log_b(a)} = n$
 $f(n) = n \rightarrow =$**

4. Thus case 2; evenly distributed

$$\begin{aligned}\text{Because } f(n) &= \Theta(n), \\ T(n) &= \Theta(n^{\log_b(a)} \log(n)) \\ &= \Theta(n^1 \log(n)) \\ &= \Theta(n \log(n))\end{aligned}$$

Master's Theorem

Let $a \geq 1$ and $b > 1$ be constants and Let $T(n)$ be defined on the non negative integers

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

Then $T(n)$ can be bounded asymptotically as follows:-

- 1) If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$
- 2) If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$
- 3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n then $T(n) = \Theta(f(n))$

Solve the recurrence relation $2T(n/2)+1$ using master's method

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

$$a=2 \quad b=2 \quad f(n)=1 \quad \text{Assume } E=1$$

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

$$n^{\log_b a} > f(n) \quad \text{case 1}$$

$$T(n) = \theta(n^{\log_b a}) = \theta(n)$$

Home Work

- Solve the recurrence relation $4T(n/2)+n$ using master's method
- Solve the recurrence relation $8T(n/2)+n$ using master's method
- Solve the recurrence relation $9T(n/3)+1$ using master's method

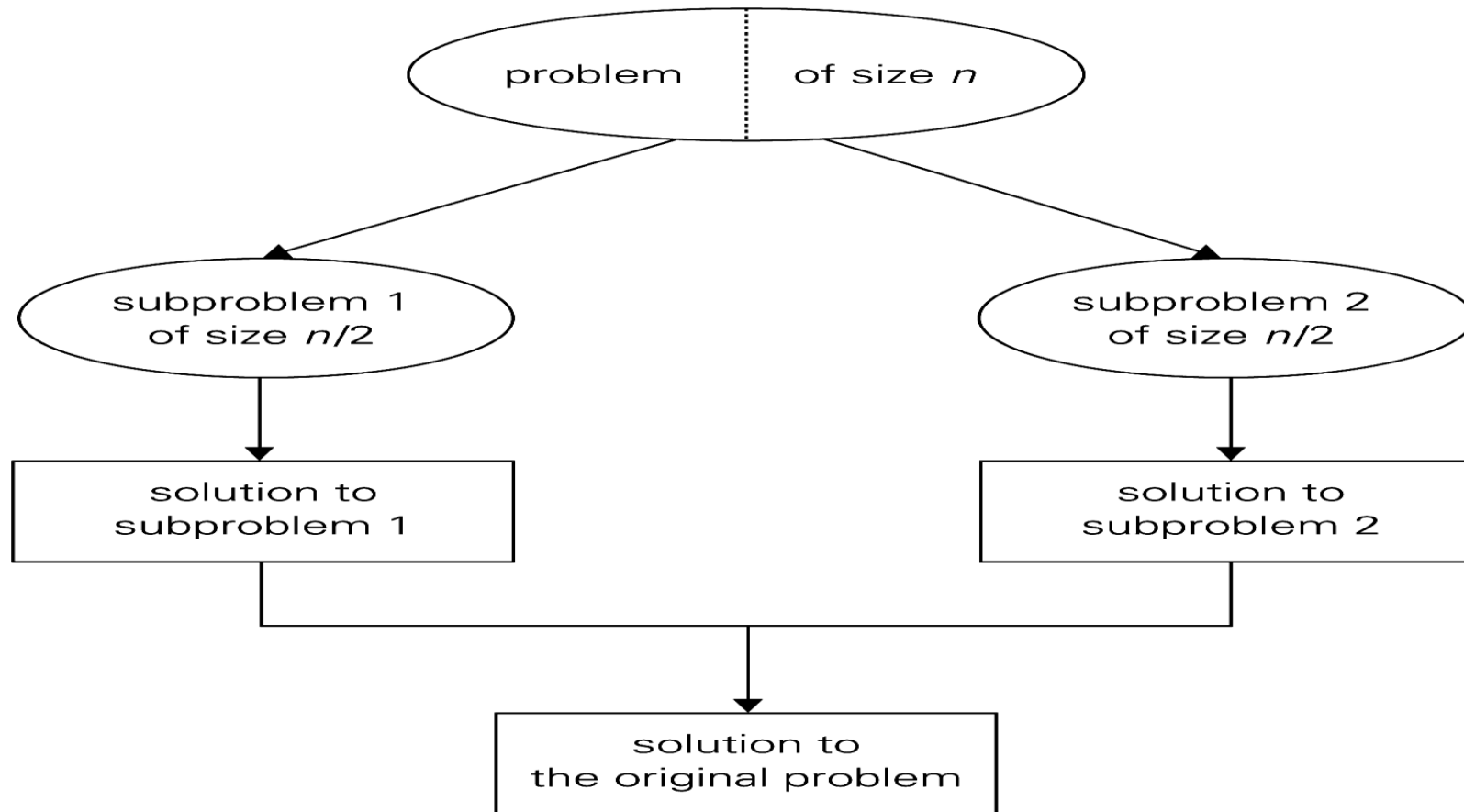
Divide and conquer method

3

- The principle behind this strategy is that it is easier to solve several small instances of a problem than one large complex problem.
- The “divide - and - conquer” technique involves in solving a particular computational problem by dividing it into smaller sub problems, solving the problem recursively and then combining the results of all the sub problems to produce the result for the original complex problem ‘P’.

The strategy involves three steps at each level of recursion.

- **Divide:-** Divide the problem into a number of sub problems.
- **Conquer:-** Conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, then just solve the sub problems in a straight forward manner.
- **Combine:-** Combine the solutions to the sub problems to form the solution for the original problem.



A typical case of divide and conquer technique

Advantages of Divide and Conquer

- The divide and conquer approach provides an efficient method for designing an algorithm in computer science because it just requires breaking of a large problem into sub problems and combining the solution of each sub problem to create the solution of the large problem.
- This technique can be broadly used to solve difficult problems because a larger problem is broken down into smaller, independent sub problems that can be easily solved.
- **Disadvantages of Divide and Conquer**
- The divide and conquer paradigm are slow in nature because of the fact that they are recursive in nature so more time is wasted in transferring the control from function call to the function definition and vice versa each time.

Divide and Conquer Strategy

7

Control Abstraction For Divide and Conquer:-

- A control abstraction shows clearly the flow of control of a procedure but the primary operations are specified by other procedures. The control abstraction can be written either iteratively or recursively.
- If we are given a problem with 'n' inputs and if it is possible for splitting the 'n' inputs into 'k' subsets where each subset represents a sub problem similar to the main problem then it can be achieved by using divide and conquer strategy.
- If the sub problems are relatively large then divide and conquer strategy is reapplied. The sub problem resulting from divide and conquer design are of the same type as the original problem. Generally divide and conquer problem is expressed using recursive formulas and functions.

A general divide and conquer design strategy(**control abstraction**) is illustrated as given below-

Algorithm DAndC (P)

[illegible]

- Small (P) is a Boolean valued function that determines whether the input size is small enough that the answer can be computed without splitting. If small (P) is true then function 'S' is invoked.
- Otherwise the problem 'P' is divided into smaller sub problems. These sub problems $P_1, P_2, P_3 \dots P_k$ are solved by recursive application of Divide-and-conquer.
- Finally the solution from k sub problems is combined to obtain the solution of the given problem.
- If the size of 'P' is 'n' and if the size of 'k' sub problems is n_1, n_2, \dots, n_k respectively then the computing time of DAndC is described by the recurrence relation.

$$T(n) = \begin{cases} g(n) & \text{when } n \text{ is small} \\ T(n_1) + T(n_1) + T(n_2) + \dots + T(n_k) + f(n) & \text{otherwise.} \end{cases}$$

$T(n)$ denotes the time for DAndC on any input of size 'n'.

$g(n)$ is the time to compute the answer directly for small inputs.

$f(n)$ is the time for dividing 'P' and combining the solutions of sub problems

A typical Divide and Conquer algorithm solves a problem using following three steps.

1. *Divide*: Break the given problem into subproblems of same type.
2. *Conquer*: Recursively solve these subproblems
3. *Combine*: Appropriately combine the answers

Following are some standard algorithms that are Divide and Conquer algorithms.

- 1) **Binary Search**
- 2) **Quicksort**
- 3) **Merge Sort**

Merge Sort

- Two methods for solving merge sort – Two – Way merge sort and Merge sort
- Two Way Merge sort – Iterative / repeating Process using loops
- Merge sort – Recursive Process
- Merging?? Process of combining two sorted list into a single sorted list.

| | A | B | C |
|----|----|--------|--------|
| | 2 | 5 | 2 |
| →i | 8 | 9 < -j | 5 |
| | 15 | 12 | 8 < -k |
| | 18 | 17 | 9 |
| | | | 12 |
| | | | 15 |
| | | | 17 |
| | | | 18 |

```

Algorithm Merge(A,B,m,n)
{
    i=1,j=1,k=1;
    While(i<=m && j<=n)
    {
        If(A[i]<B[j])
            C[k++]=A[i++];
        Else
            C[k++]=B[j++];
    }
    For(;i<=m;i++)
        C[k++]=A[i];
    For(;j<=n;j++)
        C[k++]=B[j];
}

```

| A | B | C | D |
|----|---|----|----|
| 4 | 3 | 8 | 2 |
| 6 | 5 | 10 | 4 |
| 12 | 9 | 16 | 18 |

L:2,3,4,4,5,6,8,9,10,12,16,18

4,3,8,2
4,3,8,4
4,5,8,4
6,5,8,4
6,5,8,18
6,9,8,18
12,9,8,18
12,9,10,18
12,10,18
12,16,18

A: 9 3 7 5 6 4 8 2

Assume 8 different list with single element

First_list=9

Second_list=3

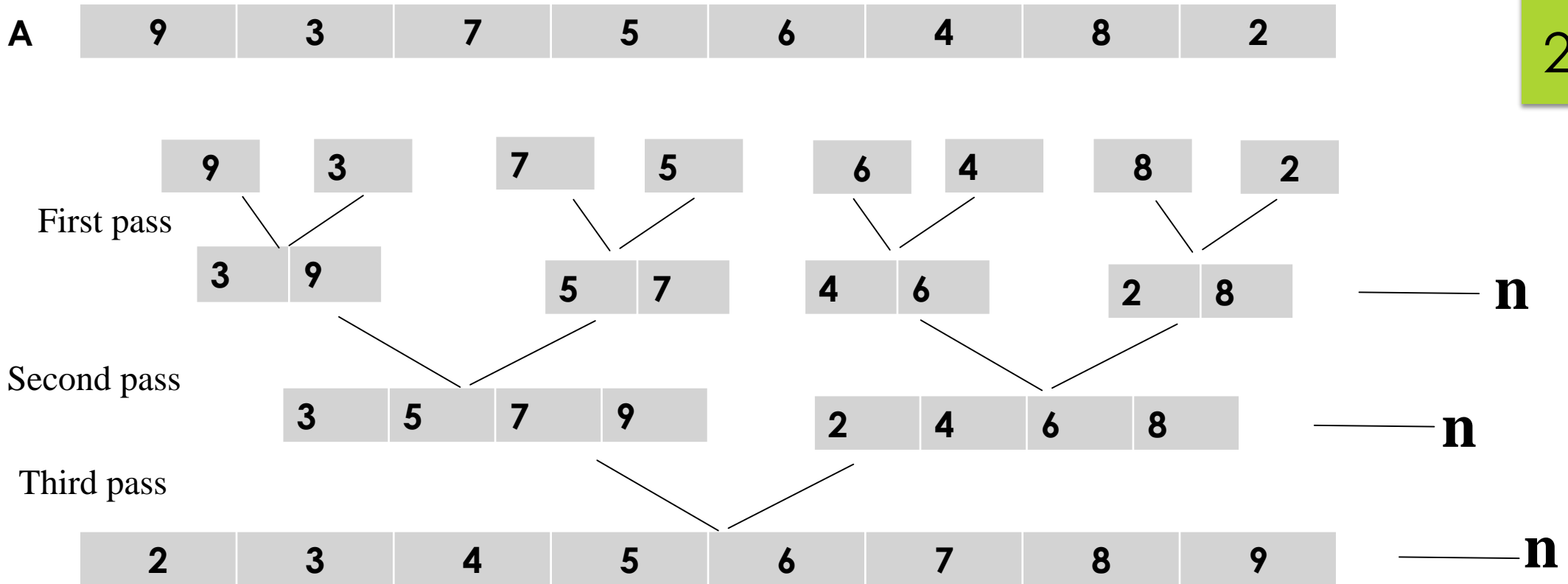
Third_list=7

.

.

.

Eighth_list=2



We are merging 2 list at a time

Number of passes= $\log n$

Total Time complexity= $\Theta(n \log n)$

$8/2/2/1 \quad 8 = 2^3$

$\log_2 8 = 3$

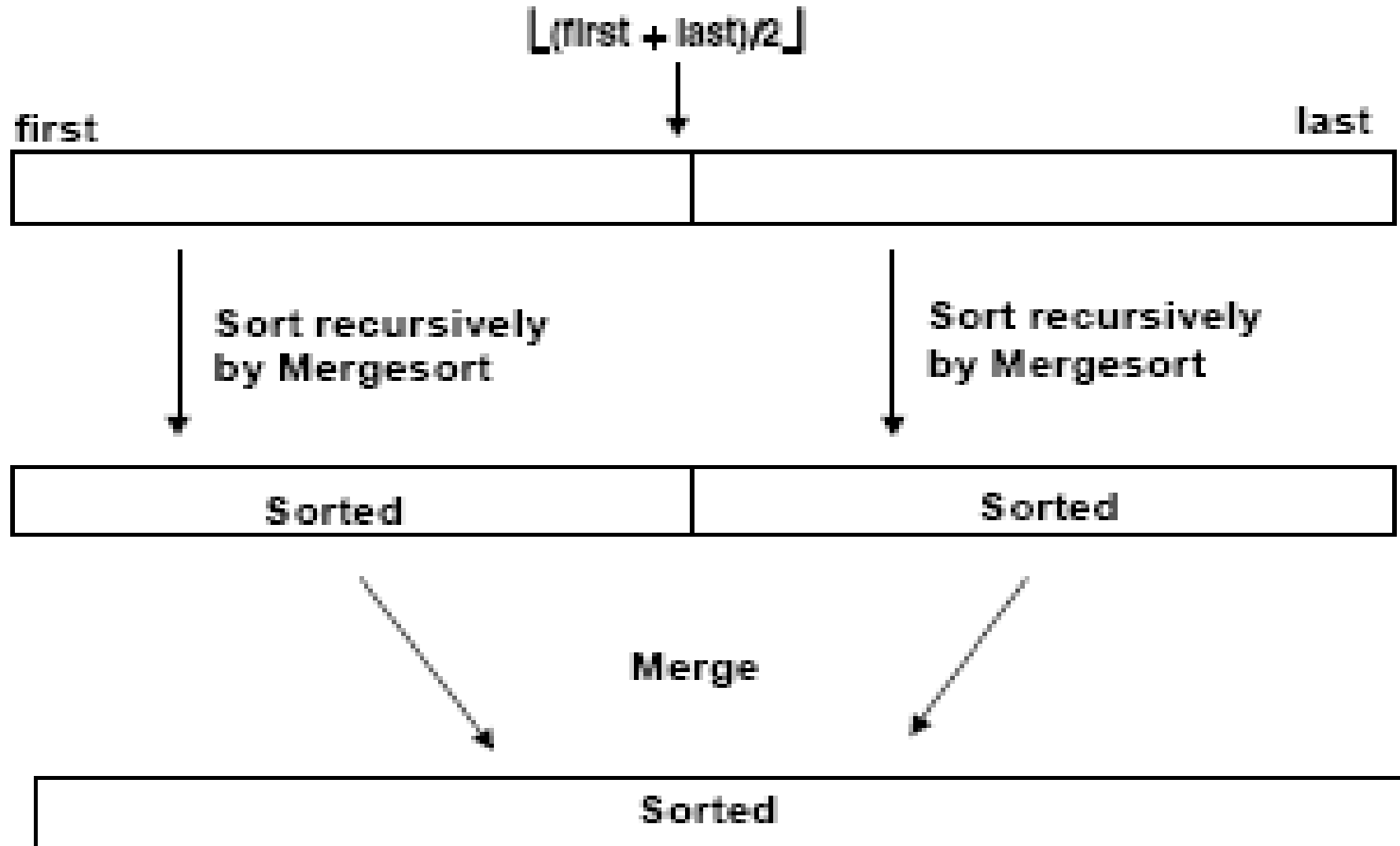
Merge Sort

25

- **Merge Sort** is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.
- The divide and conquer strategy for the merge sort divides the problem with 'n' inputs into 'k' sub problems by dividing each partition recursively in the middle until the 'p' (small) is satisfied i.e. until the size of the partition is one. Then the merge algorithm will merge the sub problems recursively until the solution for the entire problem is being obtained.
- The merge algorithm is applied on two sub lists which are in sorted order.
- **Divide:-** Divide the n-element sequence to be sorted into two sub sequences of $n/2$ elements each.
- **Conquer:-** Sort the two subsequences recursively using merge sort.
- **Combine:-** Merge the two sorted subsequences to produce the sorted answer.

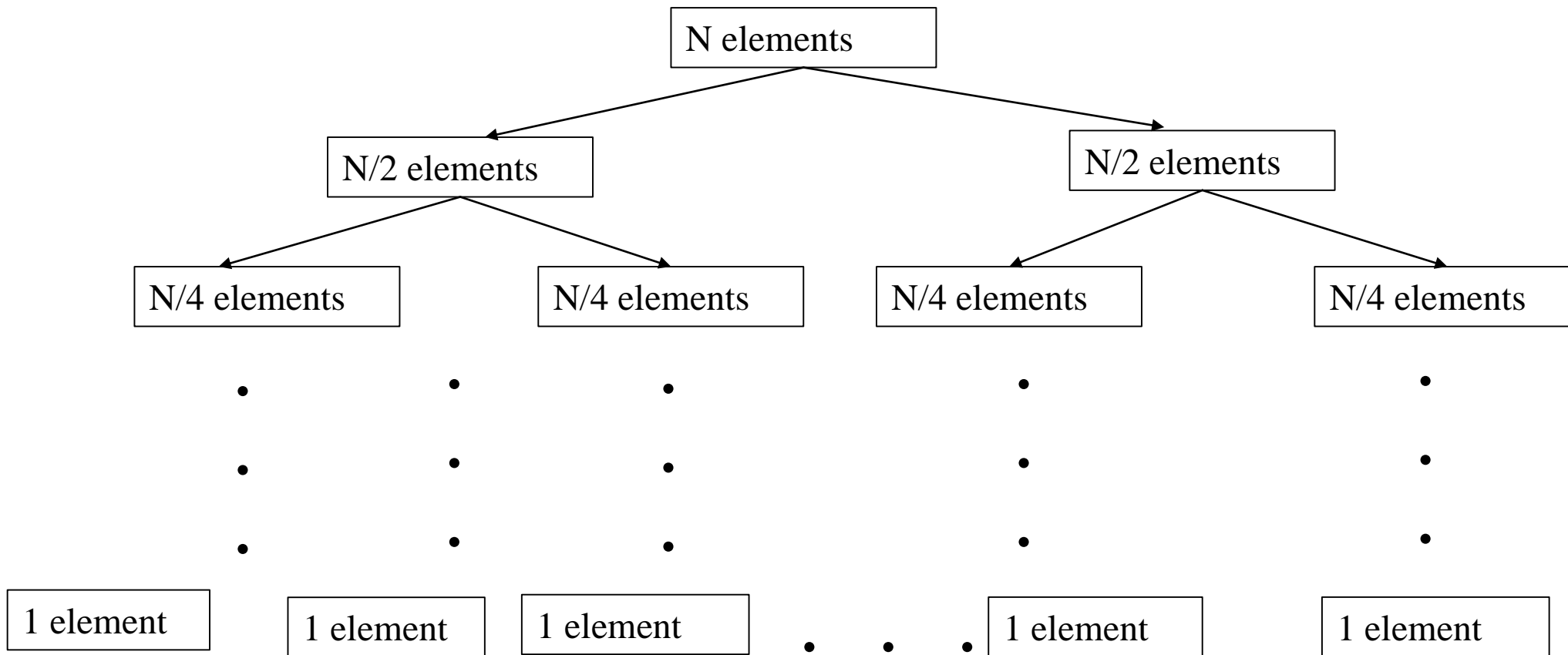
Merge Sort

26



Process of Merge Sort

27



Merge Sort - Algorithm

28

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:= [(low+high)/2];
        //Solve the subproblems
        MergeSort(low,mid);
        MergeSort(mid+1,high);
        // Combine the solutions
        Merge(low,mid,high)
    }
}
```

Algorithm Merge(low,mid,high)

//a[low:high] is a global array containing two sorted subset 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.

29

```
{
    h:=low; i:=low; j:=mid+1;
    while((h≤mid) and (j ≤ high)) do
    {
        if(a[h] ≤ a[j]) then
        {
            b[i]:=a[h];
            h=h+1;
        }
        else
        {
            b[i]:=a[j];
            j=j+1;
        }
        i=i+1;
    }
}
```



```
if(h>mid)
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];
}
```

Merge Sort- example

30

Q. Sort the given array using Merge sort.

A= 6 4 7 1 3 0 5 0

Solution

- ▶ Given an array A contains 8 elements

6 4 7 1 3 0 5 0

- ▶ The initial set of elements A[1:8] will be split in to two sets A[1:4] contain (6 4 7 1) and A[5:8] contain (3 0 5 0) after the first call to MergeSort().
- ▶ The recursive call to MergeSort() continues till we get the lists each containing only one element
- ▶ Then the Merge procedure is called, which starts sorting from these sets with one element, and moving upward in the tree, thereby sorting the entire list.

Algorithm MergeSort(l,h) -----T(n)

if(l<h)

{

Mid=l+h/2;-----1

MergeSort(1,mid);-----T(n/2)

MergeSort(mid+1,h);-----T(n/2)

Merge(l,mid,h);-----n

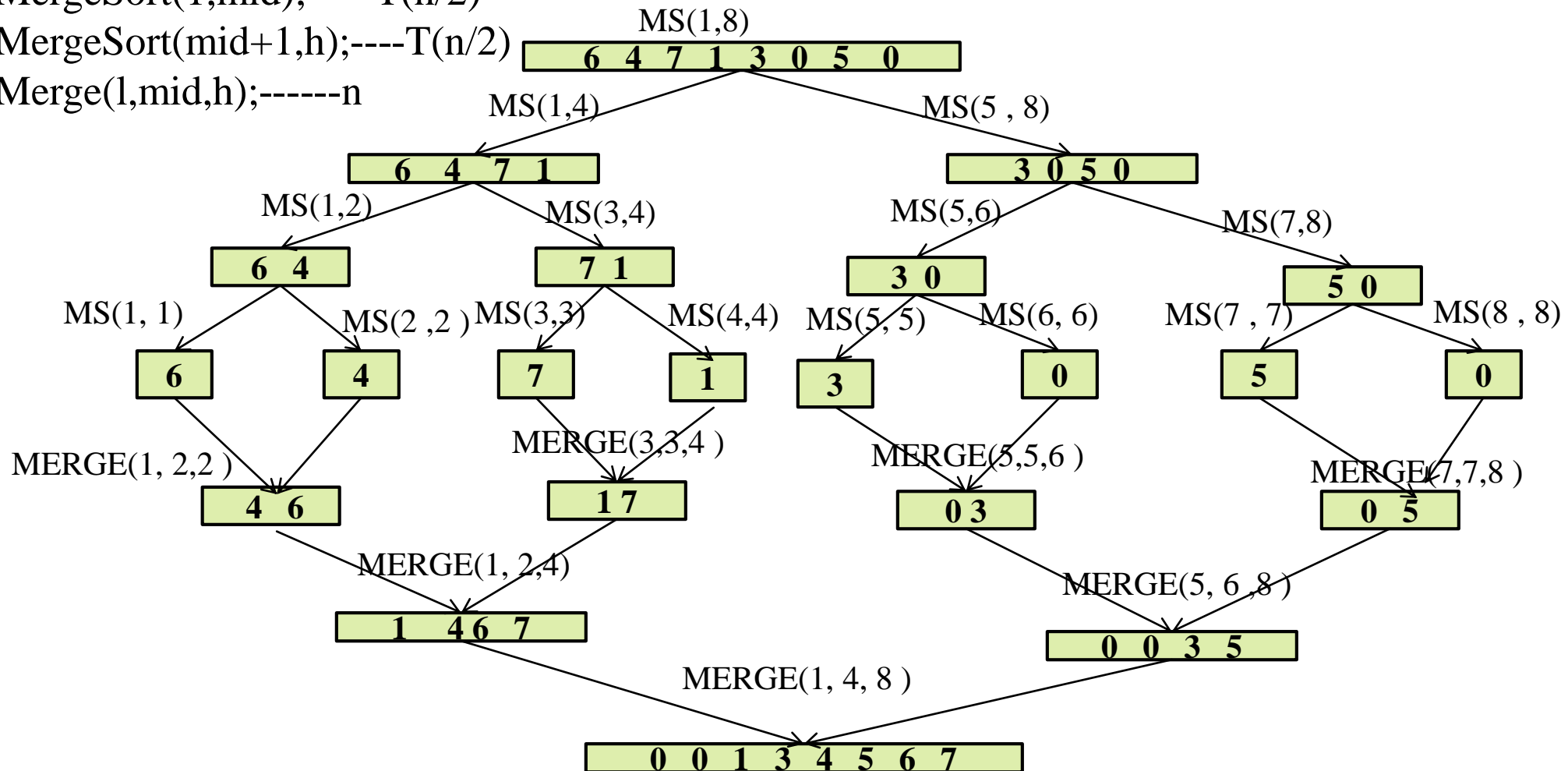
}

}

Merge Sort- example

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

31



Merge Sort

32

- Example consider the array 'a' of ten elements

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10]

310 285 179 652 351 423 861 254 450 520

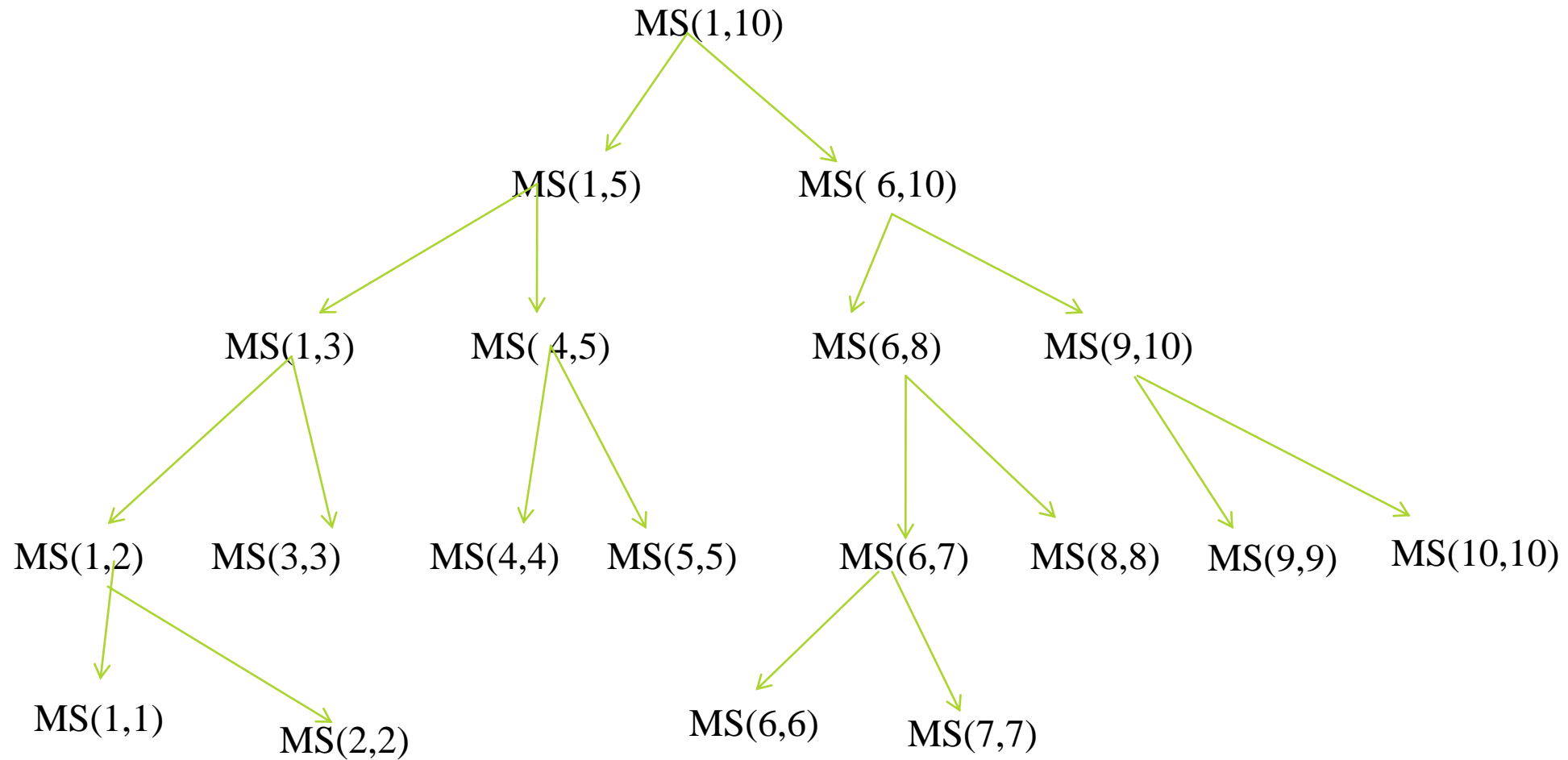
- Given an array A contains 10 elements

310 285 179 652 351 423 861 254 450 520

- The initial set of elements A[1:10] will be split in to two sets A[1:5] contain (310,285,179,652,351) and A[6:10] contain (423,861,254,450,520) after the first call to MergeSort().
- The recursive call to MergeSort() continues till we get the lists each containing only one element
- Then the Merge procedure is called, which starts sorting from these sets with one element, and moving upward in the tree, thereby sorting the entire list.

Merge Sort

33



Merge sort - Example

34

Example:1

[1] [2] [3] [4] [5] [6] [7] [8]

6 4 7 1 3 0 8 0

Example:2

[1] [2] [3] [4] [5] [6] [7] [8]

8 3 2 9 7 1 5 4

Example:3

66, 33, 40, 22, 55, 88, 60, 11, 80

Complexity analysis: Merge Sort

- **Time Complexity:**
Time complexity of Merge Sort is **$O(n \log n)$ in all 3 cases (worst, average and best)** as merge sort always divides the array in two halves and take linear time to merge two halves.
- **Auxiliary Space:** $O(n)$, because additional n locations are needed, an auxiliary array is used to execute the sorting operation
- **Algorithmic Paradigm:** Divide and Conquer
- **Sorting In Place:** No in a typical implementation
- **Stable:** Yes

Complexity analysis: Merge Sort

36

If $T(n)$ represent the time for the merging operation is proportional to n then the computing time for merge sort is ,

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

Where n is a power of 2, $n=2^k$ for some integer k .

Complexity analysis: Merge Sort

37

In the iterative substitution, or “plug-and-chug,” technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

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

$$= 2[2T(n/2^2) + c(n/2)] + cn$$

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

$$= 2^3 T(n/2^3) + 3cn$$

$$= 2^4 T(n/2^4) + 4bn$$

$$= \dots$$

$$= 2^k T(n/2^k) + kcn$$

$$T(n) = \begin{cases} 2T(n/2) + cn & n > 2 \\ a & n = 1 \end{cases}$$

Note that base, $T(n)=1$, case occurs when $2^k = n$. That is, $k = \log n$.
So,

$$T(n) = 2^k T(2^k / 2^k) + kcn$$

$$= 2^k T(1) + kcn$$

$$= na + cn \log n$$

Thus, $T(n)$ is theta($n \log n$).

Quick Sort

- **Quicksort** is a sorting algorithm.
- The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side.
- Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

Quick Sort

39

- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.
- Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.
- This algorithm is quite efficient for large-sized data sets as its *average and best case complexity are of $O(n \log n)$ and worst case complexity is $O(n^2)$, where n is the number of items.*

Quick Sort- Algorithm

40

Algorithm Quicksort(A,low,high)

```
{  
  if(low<high)  
  {  
    J=Partition(A,low,high);    // j is the position of the partitioning element  
    Quicksort(A,low,j-1);  
    Quicksort(A,j+1,high); // there is no need for combining solution, because  
                           it is an in place sorting algorithm  
  }  
}
```

Algorithm Partition(A,low,high)

// this procedure partitions the elements in to 2 lists and place the key elements in to a proper position

// low is the leftmost index and high is the rightmost index

```
{
i=low,j=high+1;
V= A[low]; A[high+1]=infinity;
  repeat
  {
    repeat
      i=i+1
    until (V<=A[i]);
    repeat
      j=j-1
    until (V>=A[j]);
    if (i<j) then interchange (A,i,j);
  } until (i>=j);
A[low]= A[j];
A[j]=V;
Return j;
}
```

Algorithm Interchange(a,i,j)

// Exchange a[i] with a[j]

```
{
  p=a[i];
  a[i]=a[j];
  a[j]=p;
}
```


Quick Sort- Example

42

Examples

[1] [2] [3] [4] [5] [6] [7] [8] [9]

65 70 75 80 85 60 55 50 45

The solution is

i

j

| | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|---|----|
| 65 | 70 | 75 | 80 | 85 | 60 | 55 | 50 | 45 | | |
| 65 | 45 | 75 | 80 | 85 | 60 | 55 | 50 | 70 | 2 | 9 |
| 65 | 45 | 50 | 80 | 85 | 60 | 55 | 75 | 70 | 3 | 8 |
| 65 | 45 | 50 | 55 | 85 | 60 | 80 | 75 | 70 | 4 | 7 |
| 65 | 45 | 50 | 55 | 85 | 60 | 80 | 75 | 70 | 5 | 6 |
| 65 | 45 | 50 | 55 | 60 | 85 | 80 | 75 | 70 | 6 | 5, |

no interchange . Because $i > j$. so interchange $A[j]$ with pivot. Then list is

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| 60 | 45 | 50 | 55 | 65 | 85 | 80 | 75 | 70 |
| 55 | 45 | 50 | 60 | 65 | 70 | 80 | 75 | 85 |
| 50 | 45 | 55 | 60 | 65 | 70 | 80 | 75 | 85 |
| 45 | 50 | 55 | 60 | 65 | 70 | 75 | 80 | 85 |

Quick Sort- Example

44

Another Example for Quick Sort

Example consider the array 'a' of ten elements

| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|
| [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|

| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 310 | 285 | 179 | 652 | 351 | 423 | 861 | 254 | 450 | 520 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

Complexity Analysis of Quick Sort

45

| Case | Number of key comparison | Time complexity | Remark |
|---|--------------------------|-----------------|--------------|
| Case 1: the key matches with the first element | $T(n)=1$ | $O(n \log n)$ | Best case |
| Case 2: Key does not exist | $T(n)=n$ | $O(n^2)$ | Worst case |
| Case 3: the Key is present at any location in the array | $T(n)=(n+1)/2$ | $O(n \log n)$ | Average case |

Complexity Analysis of Quick Sort

46

The timing analysis of QuickSort algorithm is given by the following recurrence relation,

Where

$$T(n) = \begin{cases} cn + T(n/2) + T(n/2) & , n > 1 \\ a & n = 1 \end{cases}$$

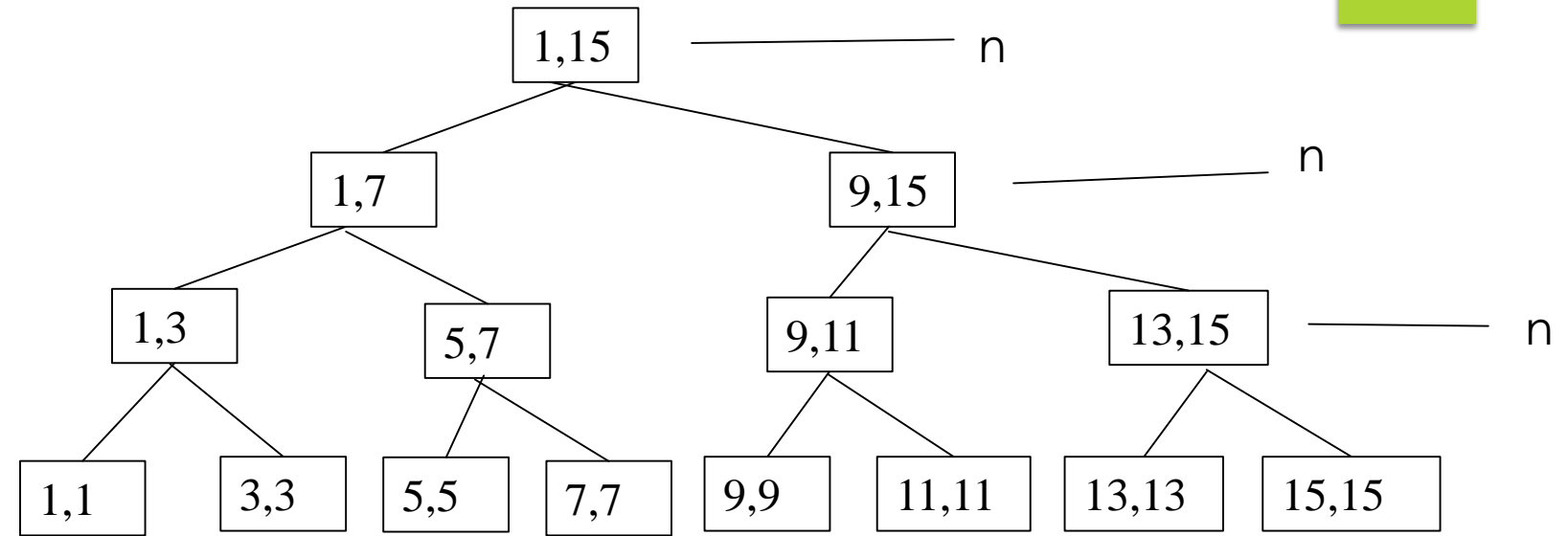
- cn is the time required for partitioning.
- $T(n/2)$ is the time required to sort the left or right subarray.
- n is a power of 2, $n = 2^k$ for some integer k .
- c and a are constant.

Analysis of Quick Sort

1,2,3.....15

8

1.....7 9.....15



Complexity Analysis of Quick Sort

48

Best case Analysis

The best case timing analysis is possible when the array is always partitioned in half. The timing analysis has taken the following form,

$$\begin{aligned}T(n) &= 2T(n/2) + cn \\&= 2[2T(n/2^2) + c(n/2)] + cn \\&= 2^2T(n/2^2) + 2cn \\&= 2^3T(n/2^3) + 3cn \\&= 2^4T(n/2^4) + 4bn \\&= \dots \\&= 2^kT(n/2^k) + kcn\end{aligned}$$

Note that base, $T(1)=a$, case occurs when $2^k=n$. That is, $k = \log n$.
So,

$$\begin{aligned}T(n) &= 2^kT(2^k/2^k) + kcn \\&= 2^kT(1) + kcn \\&= na + cn \log n\end{aligned}$$

Thus, $T(n)$ is $O(n \log n)$.

Complexity Analysis of Quick Sort

49

Worst Case Analysis

In this case on every function call the given array is partitioned into two sub arrays. One of them is an array and another one is an empty array. The timing analysis is,

$$\begin{aligned}T(n) &= cn + T(n/2) + T(n/2) \\&= cn + T(0) + T(n-1) \\&= cn + T(n-1) \\&= cn + c(n-1) + T(n-2) \\&= cn + c(n-1) + c(n-2) + \dots + c(1) + T(0) \\&= c[n + n-1 + n-2 + \dots + 3 + 2 + 1] \\&= c[n(n+1)/2] \\&= [cn^2/2] + [cn/2] \\&= O(n^2)\end{aligned}$$

Efficiency of Quicksort

50

Based on whether the partitioning is balanced.

- ▶ Best case: split in the middle — $O(n \log n)$
 - ▶ $T(n) = 2T(n/2) + \Theta(n)$ //2 subproblems of size $n/2$ each
- ▶ Worst case: sorted array! — $O(n^2)$
 - ▶ $T(n) = T(n-1) + n+1$ //2 subproblems of size 0 and $n-1$ respectively
- ▶ Average case: random arrays — $O(n \log n)$

Strassen's Matrix Multiplication Algorithm

62

Matrix multiplication

- The traditional method of matrix multiplication also known as Brute Force Method matrix multiplication.
- **Brute-force algorithm**

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$
$$= \begin{pmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{pmatrix}$$

```
Void Matrix_Mul(A,B,C)
{
    for(i=0;i<n ; i++)
    {
        for(j=0;j<n;j++)
        {
            c[i,j]=0
            for(k=0;k<n;k++)
            {
                c[i][j]+=A[i][k]*B[k][j];
            }
        }
    }
}
```

Matrix multiplication

- When multiplying two 2×2 matrices $[A_{ij}]$ and $[B_{ij}]$

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

- The resulting matrix $[C_{ij}]$ is computed by the equations

- $C_{11} = A_{11} B_{11} + A_{12} B_{21}$

- $C_{12} = A_{11} B_{12} + A_{12} B_{22}$

- $C_{21} = A_{21} B_{11} + A_{22} B_{21}$

- $C_{22} = A_{21} B_{12} + A_{22} B_{22}$

8 multiplications

4 additions

- When the size of the matrices are $n \times n$, the amount of computation is $O(n^3)$, because there are n^2 entries in the product matrix where each require $O(n)$ time to compute.

Matrix multiplication: Time Complexity

Time Complexity can be calculated as

$$i=1, j=1, k=1$$

$$T(n) = \sum_{i=1}^N \sum_{j=1}^N \sum_{k=1}^N c = Cn^3 = O(N^3)$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

The matrix multiplication is shown with red labels indicating the submatrices used in the calculation:

- A_{11} (top-left 2x2 of A), A_{12} (top-right 2x2 of A), A_{21} (bottom-left 2x2 of A), A_{22} (bottom-right 2x2 of A)
- B_{11} (top-left 2x2 of B), B_{12} (top-right 2x2 of B), B_{21} (bottom-left 2x2 of B), B_{22} (bottom-right 2x2 of B)
- C_{11} (top-left 2x2 of C), C_{12} (top-right 2x2 of C), C_{21} (bottom-left 2x2 of C), C_{22} (bottom-right 2x2 of C)

Algorithm MM(A,B,n)

```

{
  if(n≤2)
  {
    C=//4 formulas
  }
  else
  {
    Mid=n/2
    MM(A11 ,B11,n/2) +MM( A12 ,B21,n/2 )
    MM(A11 ,B12,n/2) + MM(A12 ,B22 ,n/2)
    MM(A21 ,B11 ,n/2) + MM(A22 ,B21,n/2)
    MM(A21 ,B12 , ,n/2) + MM(A22 ,B22,n/2)
  }
}

```

Strassen's Matrix Multiplication

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.

In 1969 the German mathematician Volker Strassen announced a significant more efficient algorithm for matrix multiplication of square matrix. He showed that the matrix multiplication of 2×2 matrix can be computed from just **7 multiplication and 18 additions**

Strassen's method is similar to simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size $N/2 \times N/2$, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

Strassen's Algorithm

- We can use a Divide and Conquer solution to solve matrix multiplication by separating a matrix into 4 quadrants:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

- Then we know have:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \quad C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

if $C = AB$, then we have the following:

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

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

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

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

Strassen's Algorithm

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$= \begin{pmatrix} P + S - T + V & R + T \\ Q + S & P + R - Q + U \end{pmatrix}$$

7 multiplications

18 additions / subtractions

$$P = (a_{11} + a_{22}) * (b_{11} + b_{22})$$

$$Q = (a_{21} + a_{22}) * b_{11}$$

$$R = a_{11} * (b_{12} - b_{22})$$

$$S = a_{22} * (b_{21} - b_{11})$$

$$T = (a_{11} + a_{12}) * b_{22}$$

$$U = (a_{21} - a_{11}) * (b_{11} + b_{12})$$

$$V = (a_{12} - a_{22}) * (b_{21} + b_{22})$$

Example

69

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

$$B = \begin{pmatrix} 3 & 2 \\ 2 & 2 \end{pmatrix}$$

$$P = (1+1) * 5 = 10$$

$$Q = 3 * 3 = 9$$

$$R = 1 * 0 = 0$$

$$S = 1 * -1 = -1$$

$$T = 3 * 2 = 6$$

$$U = 1 * 5 = 5$$

$$V = 1 * 4 = 4$$

$$C_{11} = 10 + (-1) - 6 + 4 = 7$$

$$C_{12} = 0 + 6 = 6$$

$$C_{21} = 9 + (-1) = 8$$

$$C_{22} = 10 + 0 - 9 + 5 = 6$$

Efficiency of Strassen's Algorithm

70

- The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} 7T(n/2) + an^2 & , n > 2 \\ b & n \leq 2 \end{cases}$$

- Where
- n is a power of 2, $n=2^k$ for some integer k . a and b are constant.

Efficiency of Strassen's Algorithm

$$\begin{aligned} T(n) &= an^2[1 + (7/4) + (7/4)^2 + \dots + (7/4)^{k-1}] + 7^k T(1) \\ &\leq cn^2 (7/4)^{\log_2 n} + 7^{\log_2 7} \\ &= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7} \\ &= O(n^{\log_2 7}) \\ &= O(n^{2.81}) \end{aligned}$$