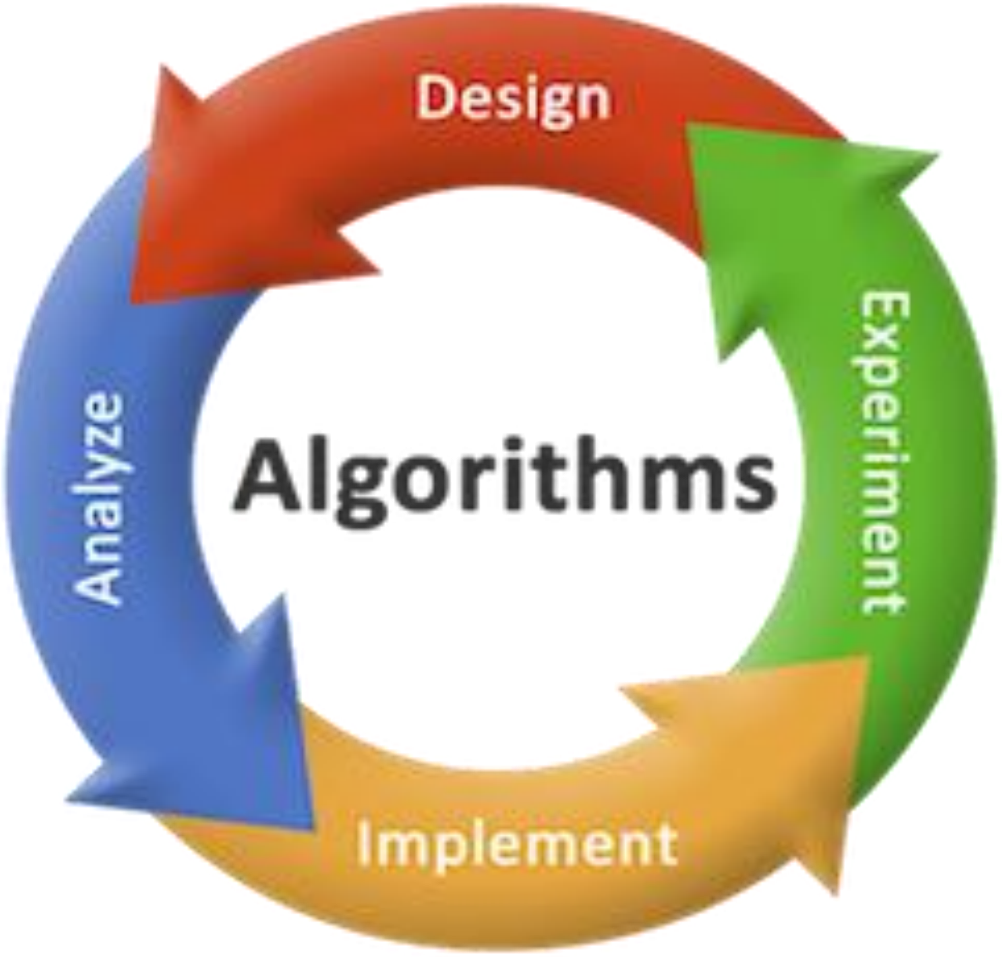
Design and analysis of algorithms

Prerequisites

- Should have basic knowledge of programming and Discrete mathematics.
- Should know the Data Structures very well.
- Should have basic understanding of Formal Language and Automata Theory

- Design and Analysis of Algorithms mainly includes



Course outcomes

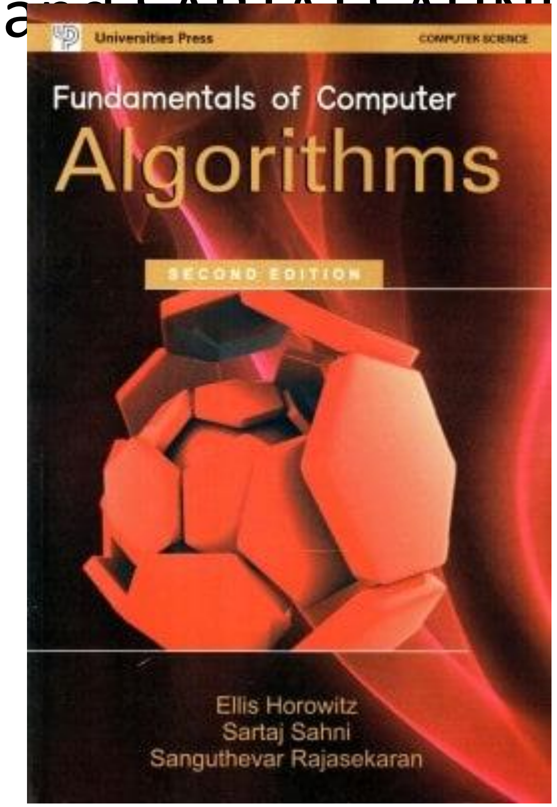
After successful completion of the course, the student will be able to:

- ❖Demonstrate asymptotic notation and Divide and Conquer Technique.(K3)
- ❖Use Greedy technique to solve various problems(K3)
- ❖Demonstrate Dynamic Programming technique to various problems(K3)
- ❖Develop algorithms using Backtracking technique.(K3)
- ❖Demonstrate Branch and Bound technique to various problems.(K3)

Text Book

Title: Fundamentals of Computer Algorithms.

Authors: ELLIS HOROWITZ a



Definition

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

All algorithms must satisfy the following criteria.

- Input: Zero or more quantities are externally supplied.
- Output: At least one quantity is produced.
- Definiteness: Each instruction is clear and unambiguous.
- Finiteness: If we trace out the instructions of an algorithm then for all cases the algorithm terminate after a finite number of steps.
- Effectiveness: Algorithm must not only definite and also feasible.

The study of algorithms mainly includes

❖ How to devise algorithms

❖ How to validate algorithms

❖ How to analyze algorithms

❖ How to test a program

BASIC TECHNIQUES FOR DESIGN OF EFFICIENT ALGORITHMS

There are basically 5 fundamental techniques which are used to design an algorithm efficiently:

- Divide-and-Conquer
- Greedy method
- Dynamic Programming
- Backtracking
- Branch-and-Bound

Design strategy	**Problems that follows**
Divide & Conquer	▪ Binary search ▪ Multiplication of two n-bits numbers ▪ Quick Sort ▪ Heap Sort ▪ Merge Sort
Greedy Method	▪ Knapsack (fractional) Problem ▪ Minimum cost Spanning tree ✓ Kruskal's algorithm ✓ Prim's algorithm ▪ Single source shortest path problem ✓ Dijkstra's algorithm
Dynamic Programming	▪ All pair shortest path-Floyed algorithm ▪ Chain matrix multiplication ▪ Longest common subsequence (LCS) ▪ 0/1 Knapsack Problem ▪ Traveling salesmen problem (TSP)
Backtracking	▪ N-queen's problem ▪ Sum-of subset
Branch & Bound	▪ Assignment problem ▪ Traveling salesmen problem (TSP)

Pseudo-Code Conventions for expressing algorithms:

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces { and }.

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

Pseudo-Code Conventions for expressing algorithms:

4. Compound data types can be formed with records. Here is an example,

```
Node= Record
{
   data type –1   data-1;
   .
   .
   .
   data type –n  data –n;
   node * link;
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with →and period.

Pseudo-Code Conventions for expressing algorithms:

5. Assignment of values to variables is done using the assignment statement.

```
<Variable>:= <expression>;
```

6. There are two Boolean values TRUE and FALSE.

❖Logical Operators	AND, OR, NOT
❖Relational Operators	<, <=,>,>=, =, !=

Pseudo-Code Conventions for expressing algorithms:

7. The following looping statements are employed.

   For, while and repeat-until

While Loop:

```
While < condition > do
{
  <statement-1>
          .
          .
  <statement-n>
}
```

Pseudo-Code Conventions for expressing algorithms:

For Loop:

```
For variable: = value-1 to value-2 step step do
{
  <statement-1>
        .
        .
        .
  <statement-n>
}
```

Pseudo-Code Conventions for expressing algorithms:

repeat-until:

```
repeat
  <statement-1>
        .
        .
        .
  <statement-n>
until<condition>
```

Pseudo-Code Conventions for expressing algorithms:

8. A conditional statement has the following forms.

```
If <condition> then <statement>
If <condition> then <statement-1>
        Else <statement-1>
```

Case statement:

```
Case
{
    :<condition-1> :<statement-1>
            .
            .
            .
    :<condition-n> :<statement-n>
    :else :<statement-n+1>
}
```

Pseudo-Code Conventions for expressing algorithms:

9. Input and output are done using the instructions read & write.

10. There is only one type of procedure: Algorithm,

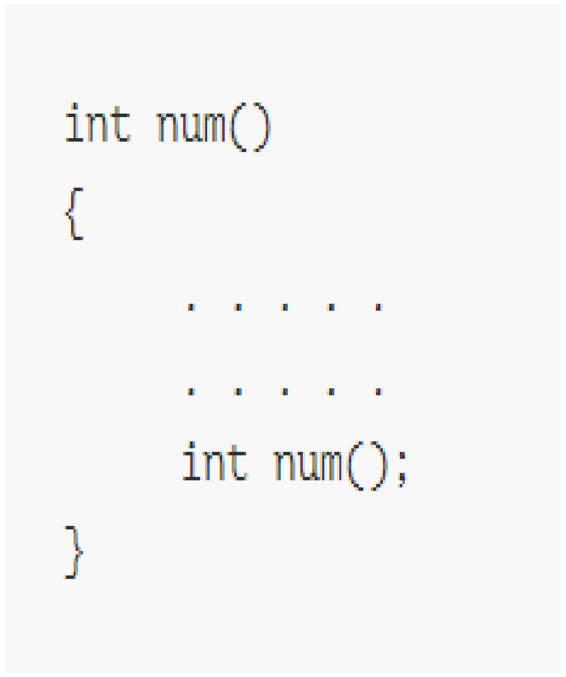
    the heading takes the form,

        Algorithm Name (Parameter lists)

Recursive algorithms

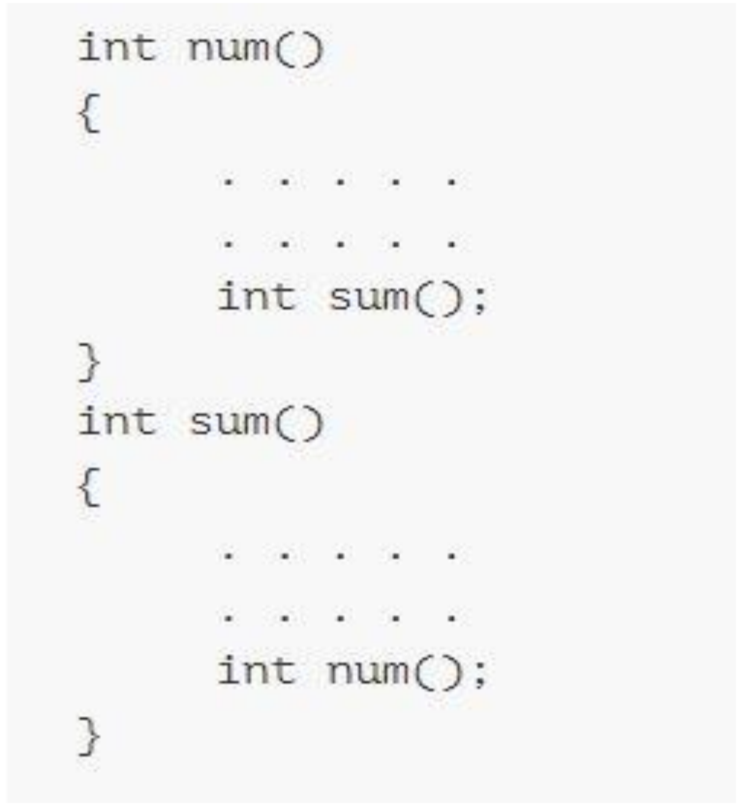
- ❖A Recursive function is a function that is defined in terms of itself.
- ❖Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.
- ❖An algorithm that calls itself is Direct Recursive.
- ❖Algorithm ‘A’ is said to be Indirect Recursive if it calls another algorithm which in turns calls ‘A’.

Direct Recursion



```
int num()
{
      . . . . .
      . . . . .
      int num();
}
```

Indirect Recursion



```
int num()
{
      . . . . .
      . . . . .
      int sum();
}
int sum()
{
      . . . . .
      . . . . .
      int num();
}
```

Factorial of Number

```
Algorithm Factorial(int n)
        {
              if n = = 1 then
                   return 1
              else
                   return factorial (n-1)*n
        }
```

Performance Analysis

Space Complexity:

The space complexity of an algorithm is the amount of memory it needs to run.

Time Complexity:

The time complexity of an algorithm is the amount of computer time it needs to run.

❖ Performance evaluation can be divided as:

- Priori Analysis
- Posteriori Analysis

Priori Analysis : It means we do analysis (space and time) of an algorithm prior to running it on a specific system.(It is independent of language of compiler and types of hardware.)

Posteriori Analysis : it means we do analysis of algorithm only after running it on a system. It directly depends on system and changes from system to system.(It is dependent on language of compiler and type of hardware.)

Space Complexity

- The Space needed by each of these algorithms is seen to be the sum of the following component.

- A fixed part - This part typically includes the instruction space (i.e., Space for the code), space for simple variable, space for constants and so on.

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, and the recursion stack space.

Space Complexity

- The space requirement s(p) of any algorithm p may therefore be written as,

  S(P) = c+ Sp(Instance characteristics)

  Where 'c' is a constant.

- When analyzing the space complexity of an algorithm, we concentrate solely on estimating Sp (instance characteristics).

- For any given problem, we need first to determine which instance characteristics to use to measure the space requirements and this is very problem specific.

Example 1

```
Algorithm abc(a,b,c)
{
  return a+b+b*c+(a+b-c)/(a+b) +4.0;
}
```

In this algorithm Sp=0;

let assume each variable occupies one word.

Then the space occupied by above algorithm is >=3.

$S(P) >= 3$

Example 2

```
Algorithm Sum(a,n)
{
  s:=0.0;
  for i:=1 to n do
        s:= s+a[i];
  return s;
}
```

In the above algorithm n, s occupies one word each and array ‘a’ occupies n number of words so S(P)>=n+3

Time Complexity

- ❖Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion.
- ❖The time T(P) taken by a program P is the sum of the compile time and the run time(execution time).
- ❖The compile time does not depend on the instance characteristics(i.e. no. of inputs, no. of outputs, magnitude of inputs, magnitude of outputs etc.) . Also we may assume that a compiled program will be run several times without recompilation .

Time Complexity

❖ We are concerned with just the runtime of a program, This runtime is denoted by t_p (instance characteristics).

❖ We can determine the number of steps needed by a program to solve a particular problem instance in one of two ways.

1. Count method
2. s/e method (steps per execution)

Count Method

- ❖In this method we introduce a variable, count into the program. This is a global variable with initial value 0.
- ❖Statements to increment count by the appropriate amount are introduced into the program. This is done so that 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= 0.0;
   count: = count+1; // count is global; it is initially zero
   for i:=1 to n do
   {
            count =count+1; // For for
            s:=s+a[i];
            count:=count+1; // For assignment
   }
   count:=count+1; // For last time of for
   count:=count+1; // For return
   return s;
} //Total of 2n+3 steps.
```

Example 2

```
Algorithm RSum(a,n)
{
    count:=count+1; // For the if conditional
    if(n<=0)then
    {
            count:=count+1;  //For the return
            return 0.0;
    }
    else
    {
            count:=count+1; //For the addition, function invocation and return
            return RSum(a,n-1)+a[n];
    }
}
```

❖When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count as

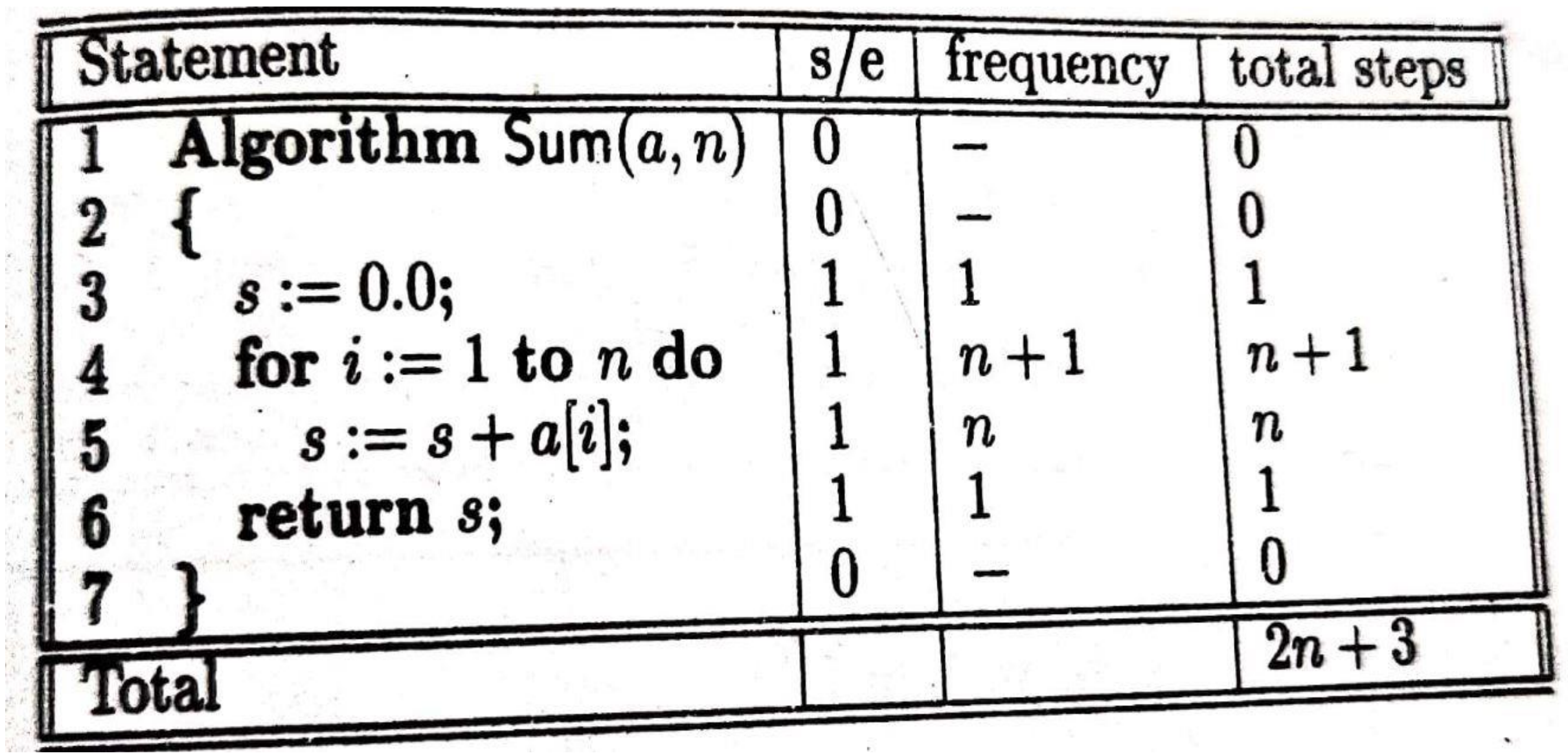
$$t_{\mathsf{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\mathsf{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

$$\begin{aligned}
t_{\mathsf{RSum}}(n) &= 2 + t_{\mathsf{RSum}}(n-1) \\
&= 2 + 2 + t_{\mathsf{RSum}}(n-2) \\
&= 2(2) + t_{\mathsf{RSum}}(n-2) \\
&\ \vdots \\
&= n(2) + t_{\mathsf{RSum}}(0) \\
&= 2n + 2, \qquad\qquad n \geq 0
\end{aligned}$$

s/e method

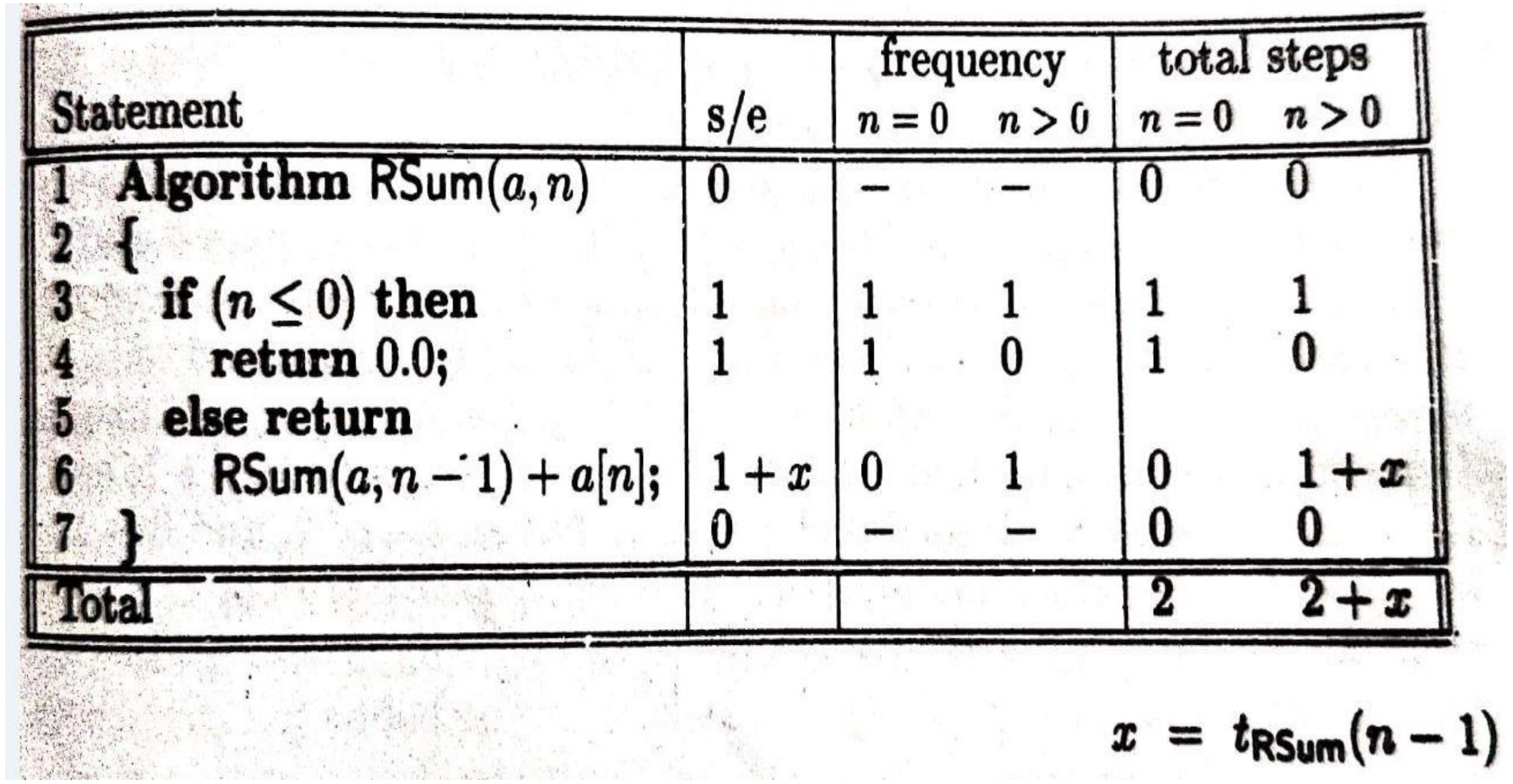
- The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.
- First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.
- By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Example 1



Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0;$	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n+1$	$n+1$
5 $s := s + a[i];$	1	n	n
6 **return** s;	1	1	1
7 }	0	–	0
Total			$2n+3$

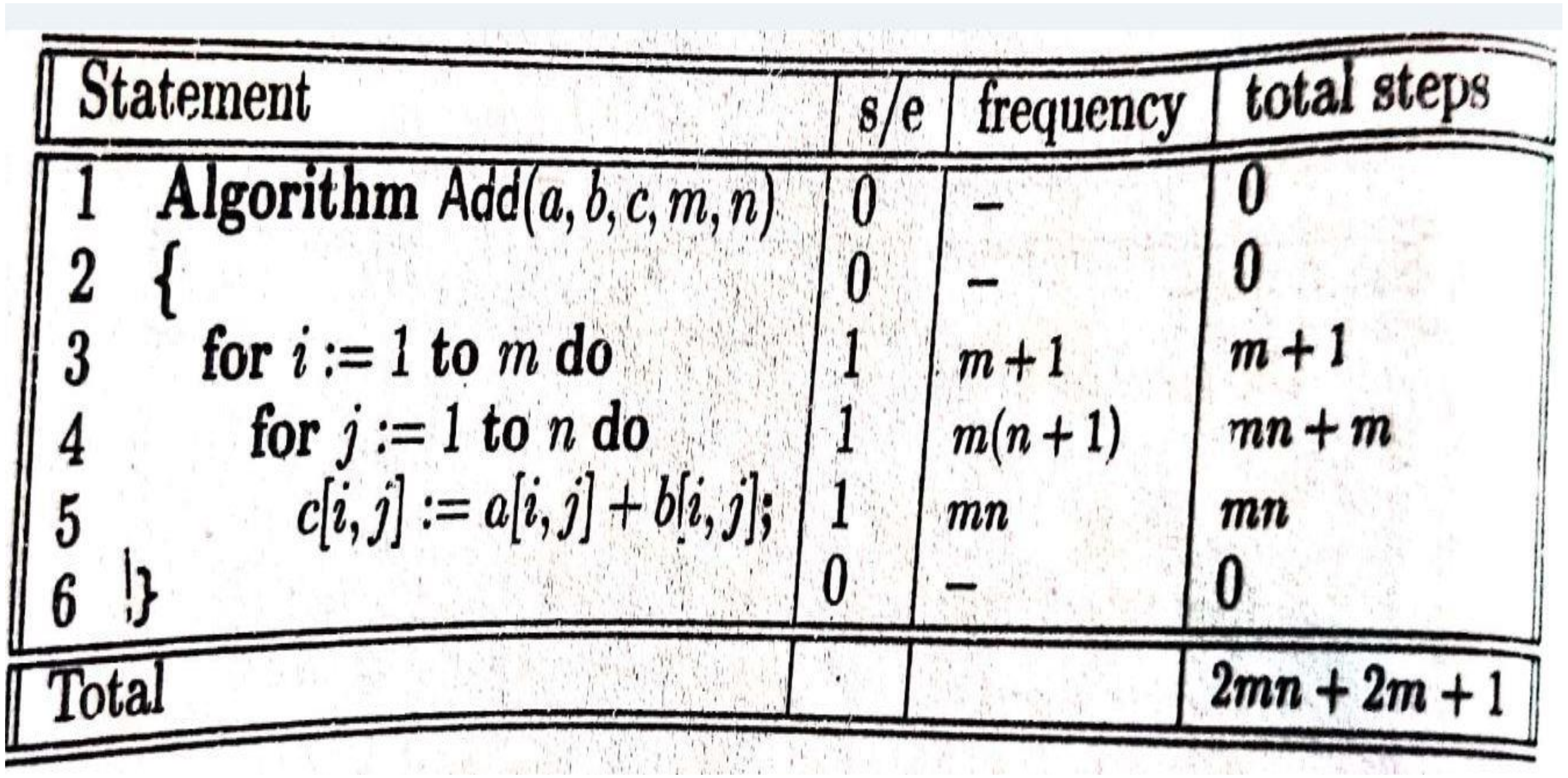
Example 2



Statement	s/e	frequency		total steps	
		$n=0$	$n>0$	$n=0$	$n>0$
1 **Algorithm** RSum(a, n)	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a, n-1) + a[n]$;	$1+x$	0	1	0	$1+x$
7 }	0	–	–	0	0
Total				2	$2+x$

$$x = t_{\mathsf{RSum}}(n-1)$$

Example 3



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 **for** $j := 1$ **to** n **do**	1	$m(n+1)$	$mn+m$
5 $c[i,j] := a[i,j] + b[i,j];$	1	mn	mn
6 }	0	–	0
Total			$2mn + 2m + 1$

Asymptotic Notations

- The asymptotic notations are used to find the time complexity of an algorithm.

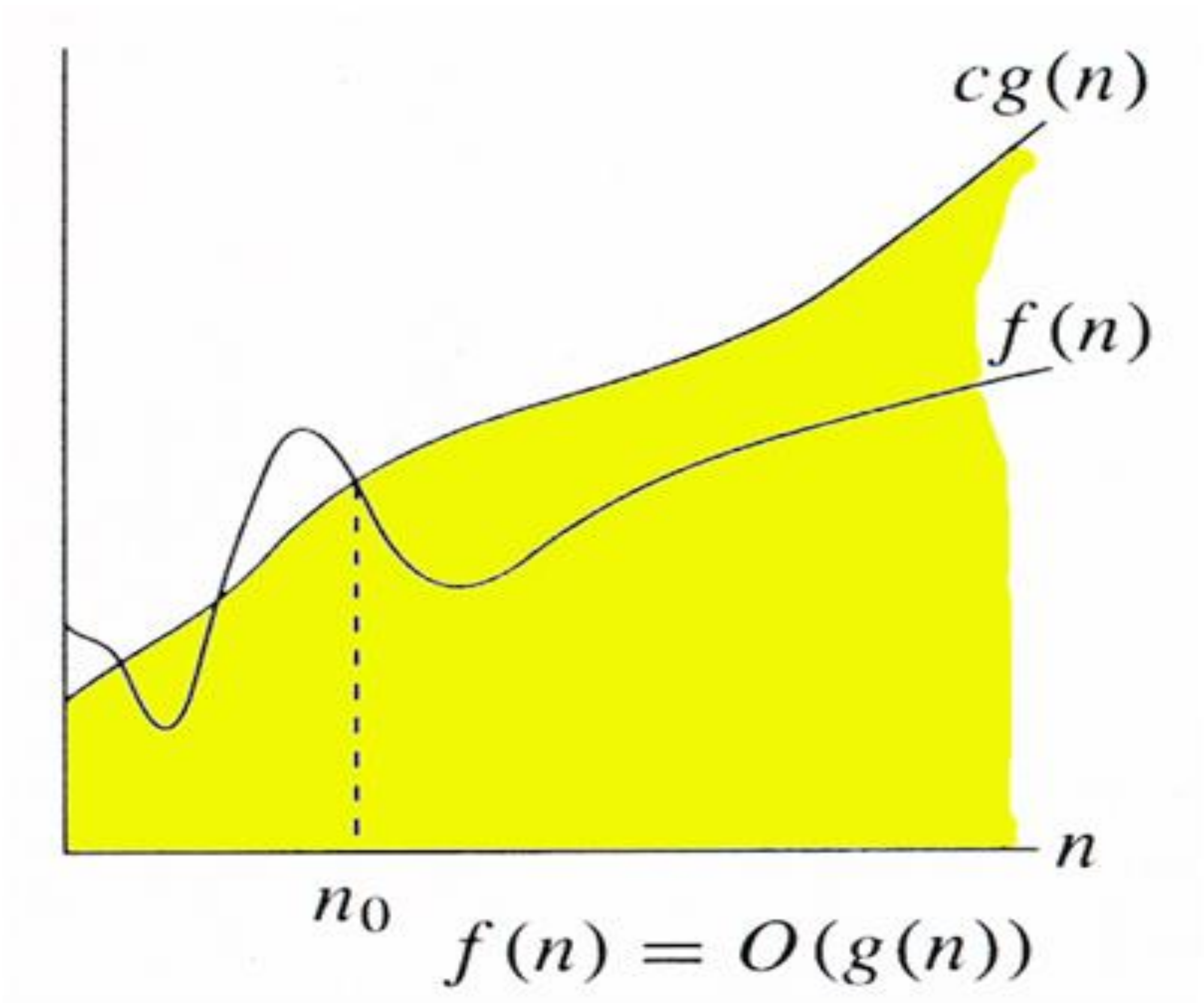
- Asymptotic notations gives fastest possible, slowest possible and average time of the algorithm.
- The basic asymptotic notations are
  - Big-oh(O),
  - Omega(Ω) and
  - theta(Θ).

BIG-OH(O)NOTATION

- ❖Big-O, commonly written as **O**, It provides us with an ***asymptotic upper bound*** for the growth rate of the runtime of an algorithm.

- ❖The function $f(n)=O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leq c*g(n)$ for all n, $n \geq n_0$

BIG-OH(O)NOTATION



Examples

1. The function $3n+2=O(n)$ as $3n+2 \leq 4n$ for all $n \geq 2$.

2. The function $10n^2+4n+2=O(n^2)$ as $10n^2+4n+2 \leq 11n^2$ for all $n \geq 5$.

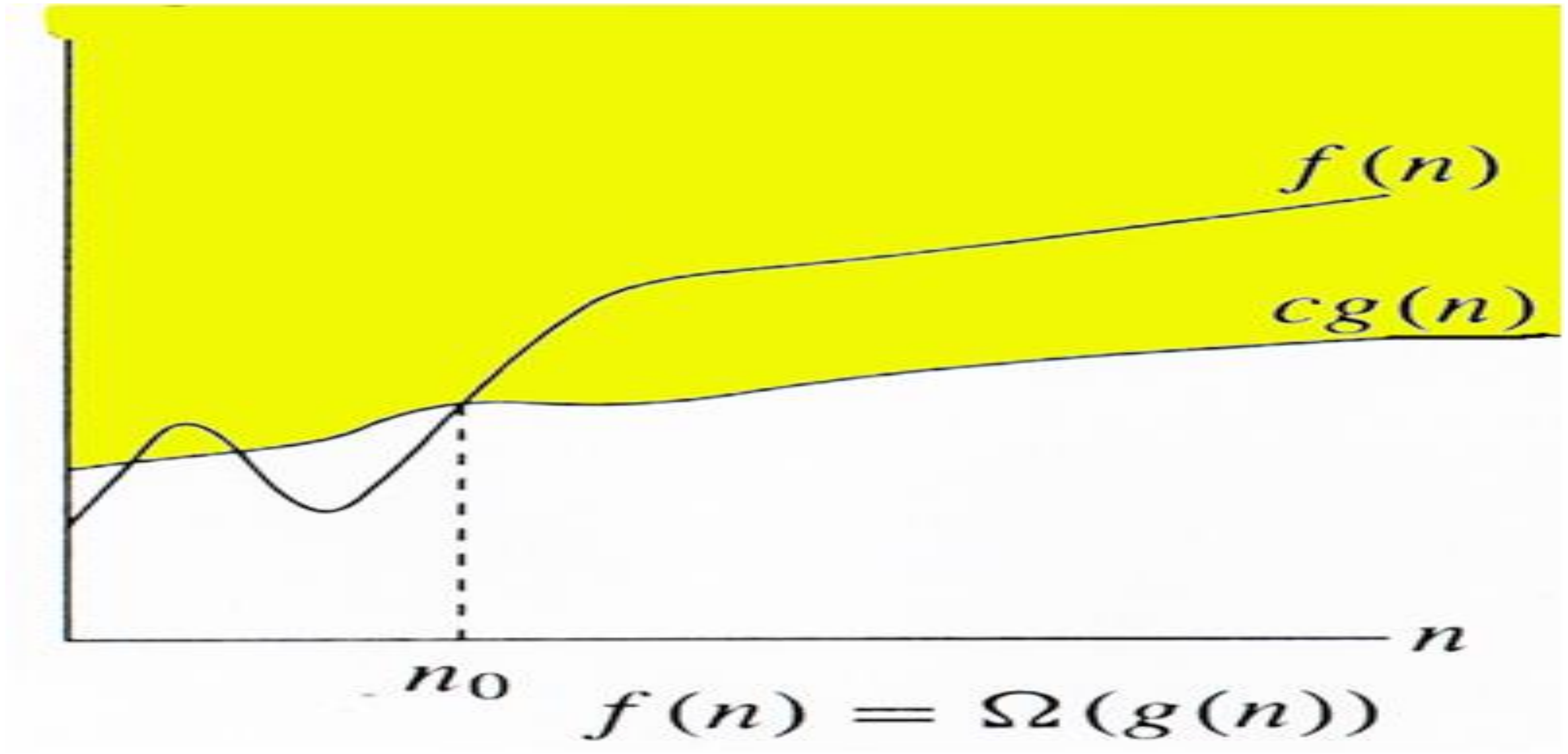
3. The function $100n^3+100n^2-6=O(n^3)$ as $1000n^3+100n^2-6 \leq 101n^2$ for all $n \geq 100$.

OMEGA(Ω)NOTATION

- Omega, commonly written as **Ω**, It provides us with an ***asymptotic lower bound*** for the growth rate of the runtime of an algorithm.

- The function f(n)= **Ω**(g(n)) iff there exist positive constants c and n_0 such that f(n) ≥ c*g(n) for all n, n≥ n_0

OMEGA(Ω)NOTATION



Example

consider $f(n)=3n+5$, $g(n)=n$

Sol : Let us assume as $c=2$

$$f(n) >= C*g(n)$$

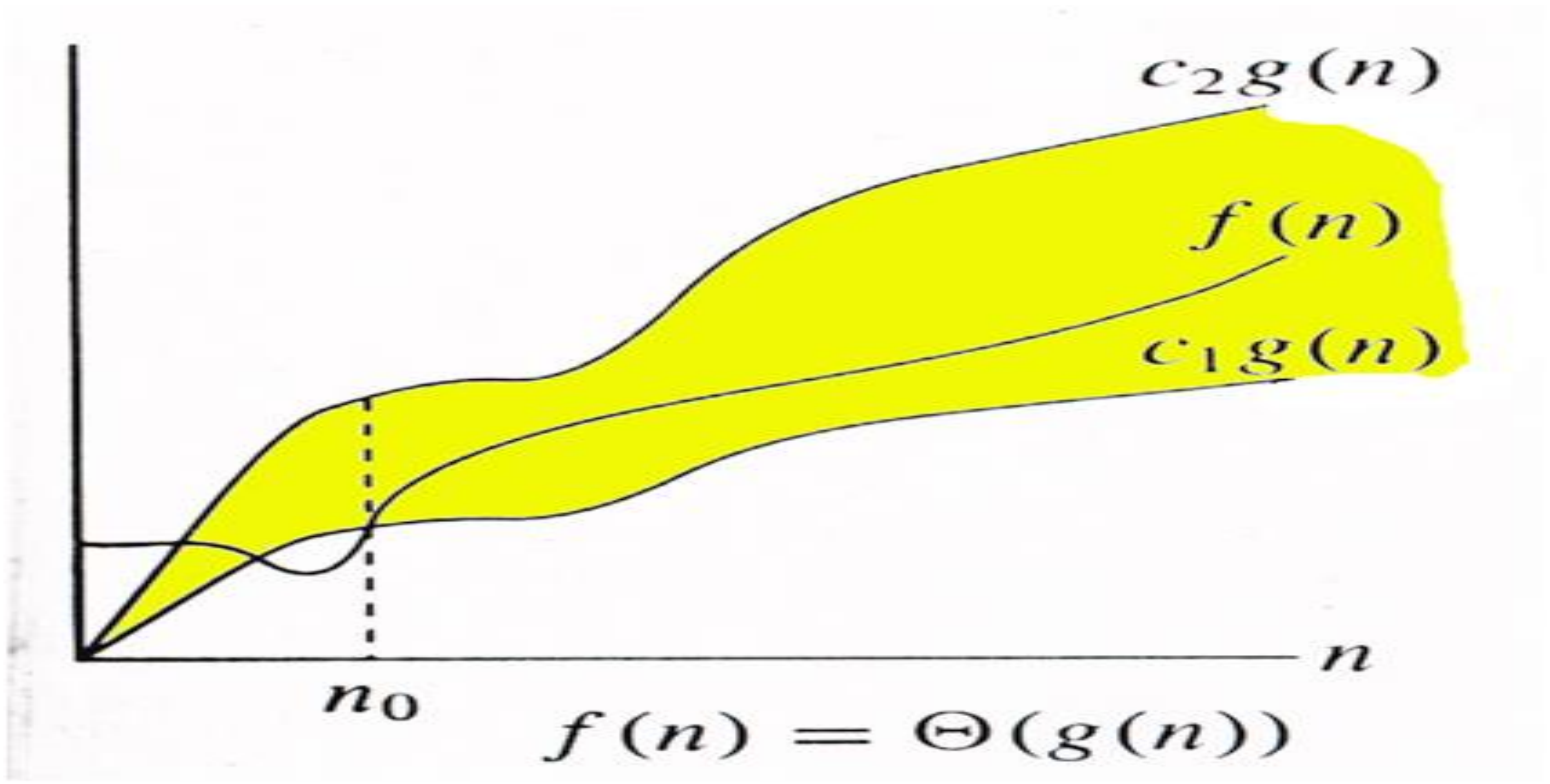
$$3n+5 >= 2n$$

for all $n>=1$, $f(n)=\Omega(n)$ i.e , $f(n)=\Omega(g(n))$

THETA(Θ)NOTATION

- ❖Theta, commonly written as **Θ**, is an Asymptotic Notation to denote the ***asymptotically tight bound*** on the growth rate of runtime of an algorithm.
- ❖The function f(n)= **Θ**(g(n)) iff there exist positive constants c_1, c_2 and n_0 such that $c_1 * g(n) \le f(n) \le c_2 * g(n)$ for all n, $n \ge n_0$

THETA(Θ)NOTATION



Example

consider $f(n)=3n+5$, $g(n)=n$

Sol :$c1*g(n)<=f(n)<=c2*g(n)$

let us assuming as $c1=2$ and $c2=4$ then

$$2n <= 3n+5 <= 4n$$

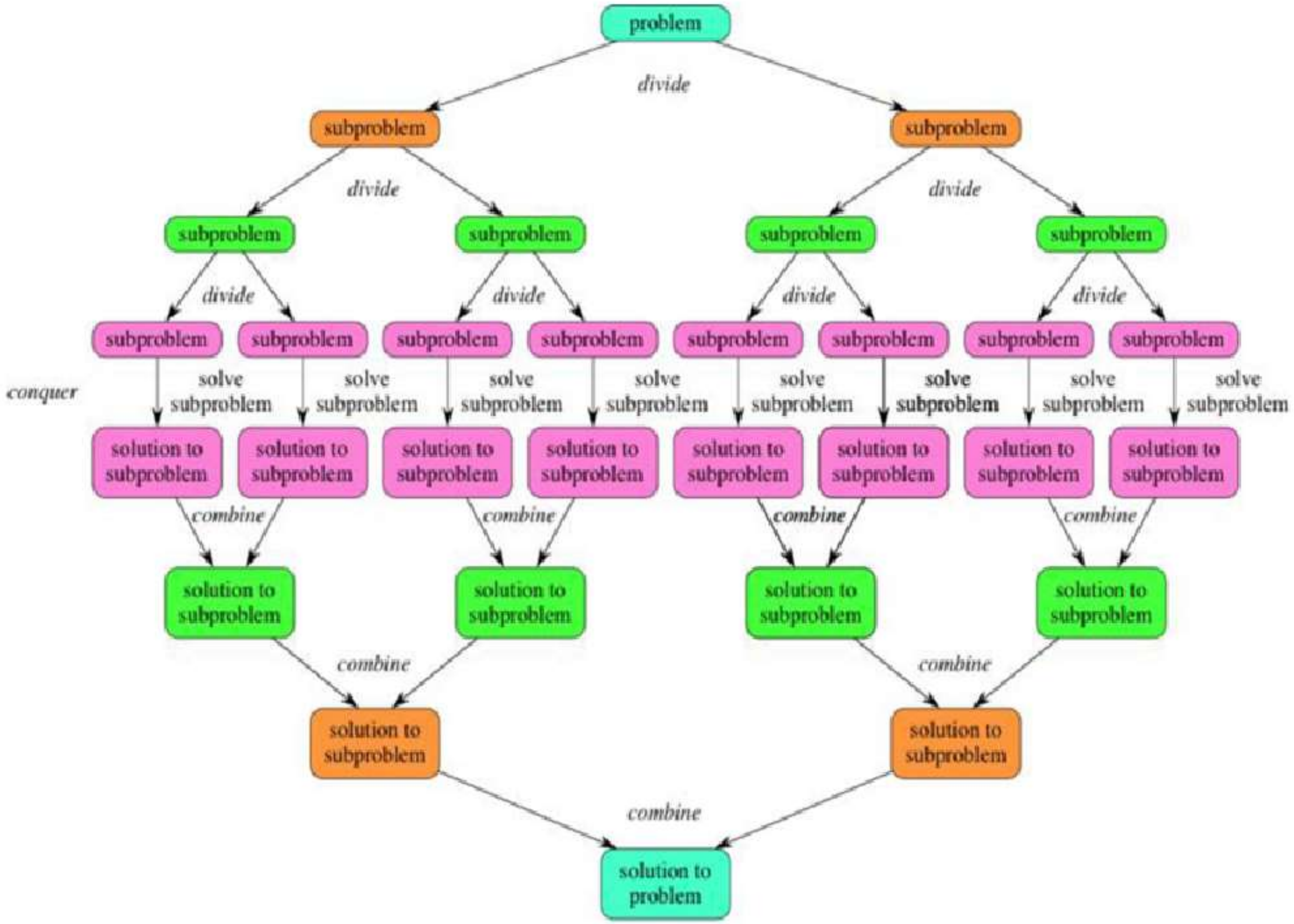
for all :$n>=3$ $f(n)=\Theta(n)$ $f(n)=\Theta (g(n))$

Performance Measurement

- Space Complexity
- Time Complexity

General Method

- Given a function to compute on n inputs the divide-and-conquer strategy Suggests splitting the inputs into k distinct subsets,1< k < n, yielding k sub problems.
- These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole. If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- Often the sub problems resulting from a divide and conquer design are of the same type as the original problem.
- For those cases the reapplication of the divide-and-conquer principle is naturally expressed by a recursive algorithm.
- Now 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.



problem
divide
subproblem
subproblem
divide
divide
subproblem
subproblem
subproblem
subproblem
divide
divide
divide
divide
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
conquer
solve subproblem
solution to subproblem
combine
solution to subproblem
combine
solution to subproblem
combine
solution to problem

```
Algorithm DAndC($P$)
{
    if Small($P$) then return S($P$);
    else
    {
        divide $P$ into smaller instances $P_1, P_2, \ldots, P_k$, $k \geq 1$;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC($P_1$),DAndC($P_2$),...,DAndC($P_k$));
    }
}
```

Control abstraction for divide-and-conquer

- DAndC is initially invoked as DAndC(P),where P is the problem to be solved.
- 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 this is so, the function S is invoked.
- Otherwise the problem P is divided into smaller sub problems. These sub problems P1,P2,………Pk are solved by recursive applications of DAndC.
- Combine is a function that determines the solution to P using the solutions to the k sub problems.If the size of P is n and the sizes of the k sub problems are n1,n2,…..nk, respectively.

- The Computing time of DAndC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases}$$

- Where T(n) is the time for DAndC on any input of size n and g(n) is the time to compute the answer directly for small inputs.
- The function f(n) is the time for dividing P and combining the solutions to sub problem.

Pros and cons of Divide and Conquer Approach

- Divide and conquer approach supports parallelism as sub problems are independent, can be solved simultaneously.
- In this approach, most of the algorithms follow recursions, very high memory is required for recursion stack.

Application of Divide and Conquer Approach

- Binary search
- Finding the maximum and minimum of a sequence of numbers
- Merge sort
- Quick Sort

1.Binary Search

- Binary Search algorithm is used for finding an item from a sorted list.
- Binary Search works by repeatedly dividing input list into two halves, where in next step we can eliminate one half which cannot contain the required key to find.
- The process repeats until we are left with only one element.
- If elements are not sorted, sort the array then apply Binary Search

```
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($a, mid + 1, l, x$);
    }
}
```

```
**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;
}
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14
$-15,$	$-6,$	$0,$	$7,$	$9,$	$23,$	$54,$	$82,$	$101,$	$112,$	$125,$	$131,$	$142,$	151

$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

At Iteration 1:

Length of array = **n**

At Iteration 2:

Length of array = **$n/2$**

At Iteration 3:

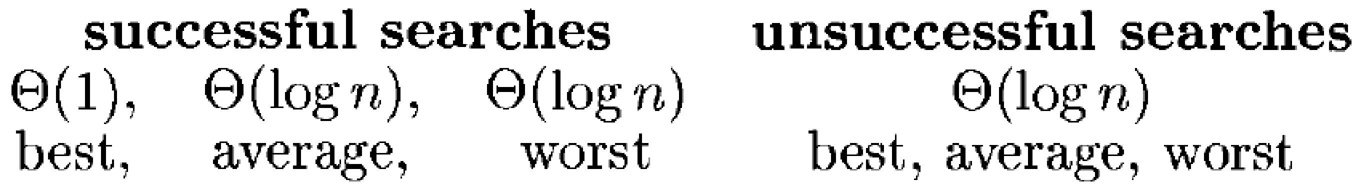
Length of array = **$(n/2)/2 = n/2^2$**

Therefore, after Iteration k:

Length of array = **$n/2^k$**

- Also, we know that after **k iterations**, the **length of the array becomes 1**
- Therefore, the Length of the array
  $n/2^k = 1$
  => **$n = 2^k$**
  Applying log function on both sides:
  => **$\log_2 n = \log_2 2^k$**
  => **$\log_2 n = k * \log_2 2$**
  As **($\log_a (a) = 1$)** Therefore, **$k = \log_2(n)$**
- **Hence, the time complexity of Binary Search is log2 (n)**

Describe the computing time of binary search by giving formulas that describe the best, average, and worst cases:



successful searches			**unsuccessful searches**
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
best,	average,	worst	best, average, worst

2.Finding maximum and minimum number

- Let P = (n, a [i],……,a [j]) denote an arbitrary instance of the problem.
- Here ‘n’ is the no. of elements in the list (a [i],….,a[j]) and we are interested in finding the maximum and minimum of the list.
- If the list has more than 2 elements, P has to be divided into smaller instances.
- For example, we might divide ‘P’ into the 2 instances, P1=([n/2],a[1],……..a[n/2]) & P2= (n-[n/2],a[[n/2]+1],….., a[n]) After having divided ‘P’ into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

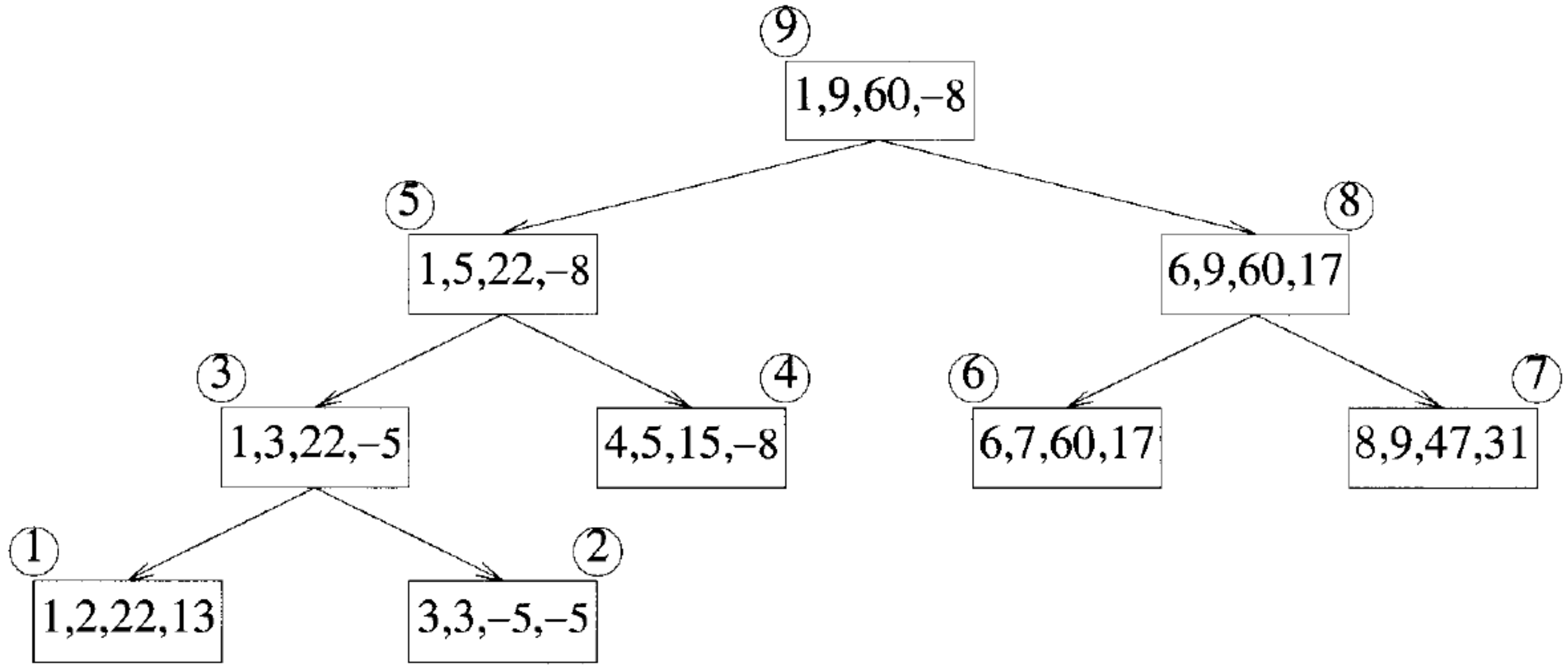
```
Algorithm StraightMaxMin($a, n, max, min$)
// Set $max$ to the maximum and $min$ to the minimum of $a[1 : n]$.
{
    $max := min := a[1]$;
    for $i := 2$ to $n$ do
    {
        if ($a[i] > max$) then $max := a[i]$;
        if ($a[i] < min$) then $min := a[i]$;
    }
}
```

```
Algorithm MaxMin($i, j, max, min$)
// $a[1:n]$ is a global array. Parameters $i$ and $j$ are integers,
// $1 \leq i \leq j \leq n$. The effect is to set $max$ and $min$ to the
// largest and smallest values in $a[i:j]$, respectively.
{
    if ($i = j$) then $max := min := a[i]$; // Small($P$)
    else if ($i = j - 1$) then  // Another case of Small($P$)
        {
            if ($a[i] < a[j]$) then
            {
                $max := a[j]$; $min := a[i]$;
            }
            else
            {
                $max := a[i]$; $min := a[j]$;
            }
        }
        else
        {   // If $P$ is not small, divide $P$ into subproblems.
            // Find where to split the set.
                $mid := \lfloor (i + j)/2 \rfloor$;
            // Solve the subproblems.
                MaxMin($i, mid, max, min$);
                MaxMin($mid + 1, j, max1, min1$);
            // Combine the solutions.
                if ($max < max1$) then $max := max1$;
                if ($min > min1$) then $min := min1$;
        }
}
```

- The procedure is initially invoked by the statement MaxMin(1,n,x,y). for this algorithm each node has four items of information: i, j, max, min. Suppose we simulate MaxMin on the following nine elements:

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

22 13 -5 -8 15 60 17 31 47



- As shown in figure, in this Algorithm, each node has 4 items of information: i, j, max & min.
- In figure, root node contains 1 & 9 as the values of i& j corresponding to the initial call to MaxMin.
- This execution produces 2 new calls to MaxMin, where i& j have the values 1, 5 & 6, 9 respectively & thus split the set into 2 subsets of approximately the same size.
- Maximum depth of recursion is 4.
- The upper left corner of each node represent the orders in which max and min are assigned values.

the number of element comparisons needed for MaxMin?If T(n) represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

T(n) = 2T(n/2) + 2, if n > 2 Let us solve this equation.
T(n) = 2T(n/2) + 2 **… (1)**
By substituting n by (n / 2) in Equation (1)
T(n/2) = 2T(n/4) + 2
⇒ T(n) = 2(2T(n/4) + 2) + 2
= 4T(n/4) + 4 + 2 **… (2)**
By substituting n by n/4 in Equation (1),
T(n/4) = 2T(n/8) + 2
Substitute it in Equation (2),
T(n) = 4[2T(n/8) + 2] + 4 + 2
= 8T(n/8) + 8 + 4 + 2
= $2^3 T(n/2^3) + 2^3 + 2^2 + 2^1$

After k – 1 iterations

$$\therefore\ T(n) = 2^{k-1}\,T\left(\frac{n}{2^{k-1}}\right) + 2^{k-1} + 2^{k-2} + \dots$$

$$+ 2^3 + 2^2 + 2^1$$

$$= 2^{k-1}\,T\left(\frac{n}{2^{k-1}}\right) + \sum_{i=1}^{k-1} 2^i$$

$$= 2^{k-1}\,T\left(\frac{n}{2^{k-1}}\right) + (2^k - 2)$$

$$\text{Let } n = 2^k \Rightarrow 2^{k-1} = \left(\frac{n}{2}\right)$$

$$\therefore\ T(n) = \left(\frac{n}{2}\right) T\left(\frac{2^k}{2^{k-1}}\right) + (n-2)$$

$$= \left(\frac{n}{2}\right) T(2) + (n-2)$$

For $n = 2$, $T(n) = 1$ (two elements require only 1 comparison to determine min-max)

$$T(n) = \left(\frac{n}{2}\right) + (n-2) = \left(\frac{3n}{2} - 2\right)$$

3. Merge sort

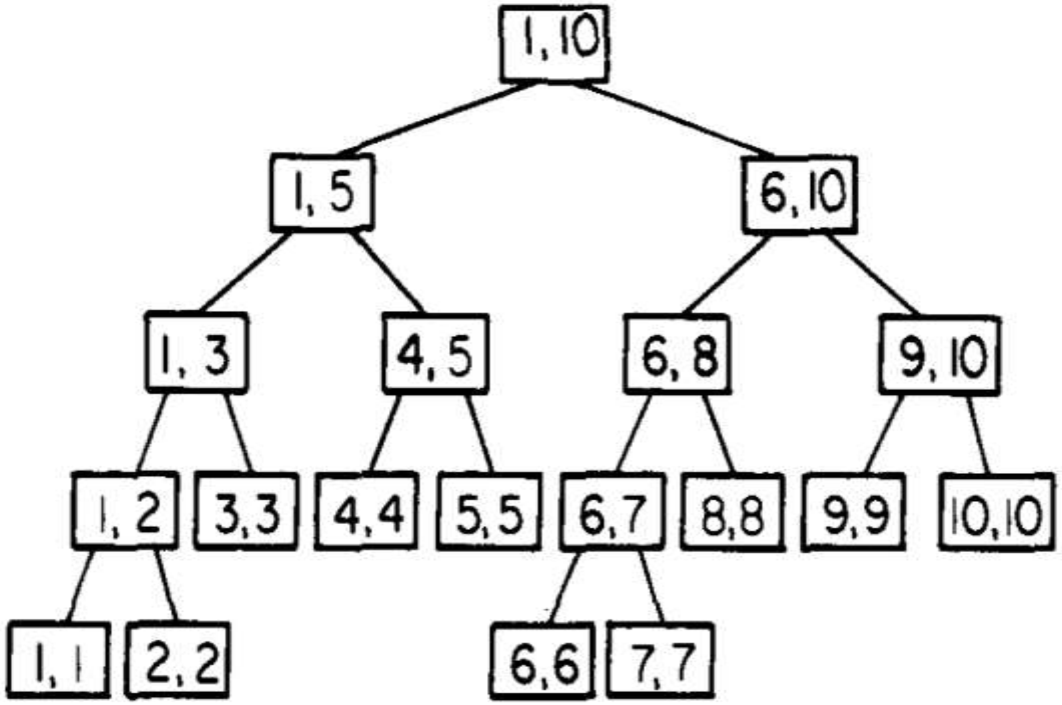
- As another example of divide-and-conquer, the worst case, best case, average case time complexity is 0(nlogn).
- Given a sequence of n elements(also called keys) a[1],,.a[n] the general idea is to imagine them split into two sets a[1],,.a[n/2]and a[n/2+1],,a[n]
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of n elements.
- We implemented this by using two algorithms MergeSort describes this process very succinctly using recursion and a Merge which merges two sorted sets.
- Then MergeSort(1,n) causes the keys to be rearranged into non-decreasing order in a.

```
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($low, mid, high$);
    }
}
```

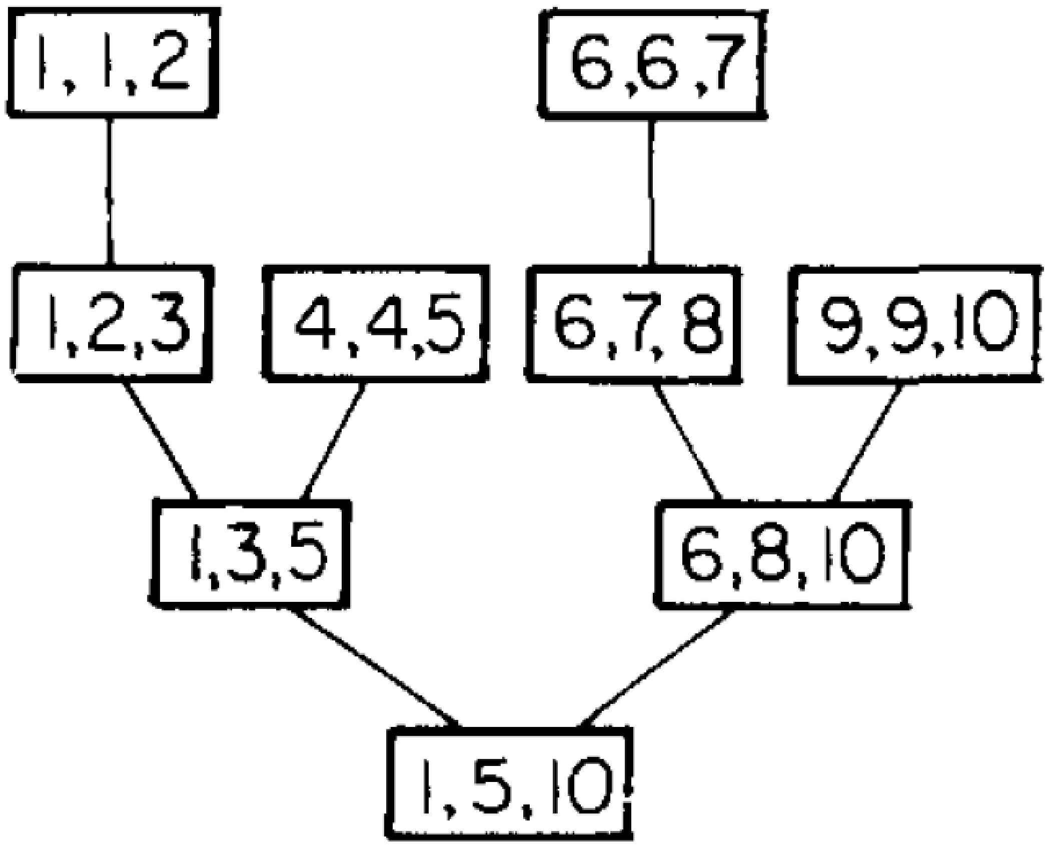
```
Algorithm Merge($low, mid, high$)
// $a[low : high]$ is a global array containing two sorted
// subsets in $a[low : mid]$ and in $a[mid + 1 : high]$. The goal
// is to merge these two sets into a single set residing
// in $a[low : high]$. $b[\,]$ is an auxiliary global array.
{
    $h := low$; $i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

Example and tree of calls to merge sort

Consider the array of ten elements *A = (310, 285, 179, 652, 351, 423,*861, 254, 450, 520).



Tree of calls to merge



If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

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

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$

It is easy to see that if $2^k < n \le 2^{k+1}$, then $T(n) \le T(2^{k+1})$. Therefore

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

4.Quick sort

- In merge sort, the list a[n] was divided at its mid point into sub arrays which were independently sorted and later merged.
- In quick sort, the division into two sub arrays is made so that the sorted sub arrays do not need to be merged later.
- This is accomplished by Rearranging the elements in a[1:n] such that a[i] < a[j] for all i between 1 and m and all j between m + 1 and n for some m, 1< m < n.
- Thus, the elements in a[1:m] and a[m+ 1:n] can be independently sorted. No merge is needed.
- The re arrangement of the elements is accomplished by picking Some element of a[ ],say t = a[s],and then reordering the other elements So that all elements appearing before t in a[1:n] are less than or equal to t and all elements appearing after t are greater than or equal to t.
- This Rearranging is referred to as partitioning.

```
**Algorithm** QuickSort($p, q$)
// Sorts the elements $a[p], \ldots, a[q]$ which reside in the global
// array $a[1:n]$ into ascending order; $a[n+1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1:n]$.
{
    if ($p < q$) then  // If there are more than one element
    {
        // divide $P$ into two subproblems.
            $j :=$ Partition($a, p, q+1$);
                // $j$ is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort($p, j-1$);
            QuickSort($j+1, q$);
        // There is no need for combining solutions.
    }
}
```

```
**Algorithm** Partition($a, m, p$)
// Within $a[m], a[m+1], \ldots, a[p-1]$ the elements are
// rearranged in such a manner that if initially $t = a[m]$,
// then after completion $a[q] = t$ for some $q$ between $m$
// and $p-1$, $a[k] \leq t$ for $m \leq k < q$, and $a[k] \geq t$
// for $q < k < p$. $q$ is returned. Set $a[p] = \infty$.
{
    $v := a[m]$; $i := m$; $j := p$;
    **repeat**
    {
        **repeat**
            $i := i + 1$;
        **until** $(a[i] \geq v)$;

        **repeat**
            $j := j - 1$;
        **until** $(a[j] \leq v)$;

        **if** $(i < j)$ **then** Interchange($a, i, j$);

    } **until** $(i \geq j)$;

    $a[m] := a[j]$; $a[j] := v$; **return** $j$;
}
```

```
**Algorithm** Interchange($a, i, j$)
// Exchange $a[i]$ with $a[j]$.
{
    $p := a[i]$;
    $a[i] := a[j]$; $a[j] := p$;
}
```

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

Time complexity of quick sort

- The worst case complexity is $O(n^2)$
- The average case and best case complexity is $O(n\log n)$.

Performance Measurements

- QuickSort and MergeSort were evaluated on a SUN workstation10/30.
- In both cases the recursive versions were used.
- For QuickSort the Partition function was altered to carry out the median of three rule (i.e. the partitioning element was the median of (a[m],a[[(m+p-1)/2] and a[p-1]).
- Each data set consisted of random integers in the range (0, 1000). Tables 3.5 and 3.6 record the actual computing times in milli seconds. Table 3.5 displays the average computing times.For each n=50 random data sets were used. Table 3.6 shows the worst-case computing times for the 50 data sets.
- Scanning the tables,we immediately see that QuickSort is faster than MergeSort for all values. Even though both algorithms require 0(nlogn) time on the average,QuickSort usually performs well in practice.

n	1000	2000	3000	4000	5000
MergeSort	72.8	167.2	275.1	378.5	500.6
QuickSort	36.6	85.1	138.9	205.7	269.0
n	6000	7000	8000	9000	10000
MergeSort	607.6	723.4	811.5	949.2	1073.6
QuickSort	339.4	411.0	487.7	556.3	645.2

Table 3.5 Average computing times for two sorting algorithms on random inputs

n	1000	2000	3000	4000	5000
MergeSort	105.7	206.4	335.2	422.1	589.9
QuickSort	41.6	97.1	158.6	244.9	397.8
n	6000	7000	8000	9000	10000
MergeSort	691.3	794.8	889.5	1067.2	1167.6
QuickSort	383.8	497.3	569.9	616.2	738.1

Table 3.6 Worst-case computing times for two sorting algorithms on random inputs