Introduction To Algorithm and Time Complexity

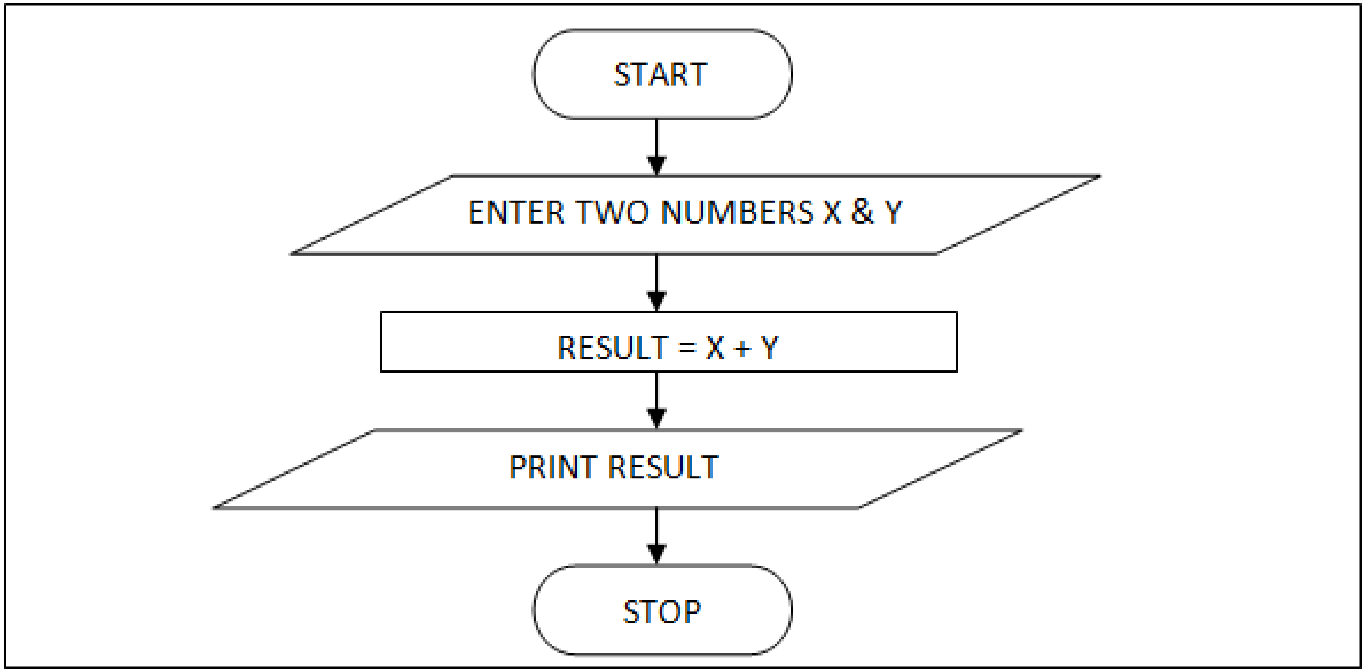
By Yash Gupta

Algorithm

- Input
- Output
- Computation
- The following are the methods to write an algorithm
  - Flow-chart
  - Pseudocode

Flow-Chart

- A graphical approach to demonstrate the logic of the code



Pseudocode

Format:

Algorithm algoName([Parameters])

Purpose Statement

Precondition Statements

PostCondition Statements

Return Statement

Sequence | Selection | Loop Statements

    Sequence | Selection | Loop Statements

end Algorithm

Example : Algorithm to swap two memory locations

Algorithm swapLocations(source, dest)

Swaps the content of two memory locations

    Pre : source and dest contains the element

    Post : Values of source and dest are exchanged

    Return : True if swapped else false

```
set temp = source
set source = dest
set dest = temp
return true
end swapLocations
```

Algorithm Efficiency

- Critical resources
  - Time
  - Space
- Purpose
  - Compare algorithms
  - Execution Speed
  - Minimizing space

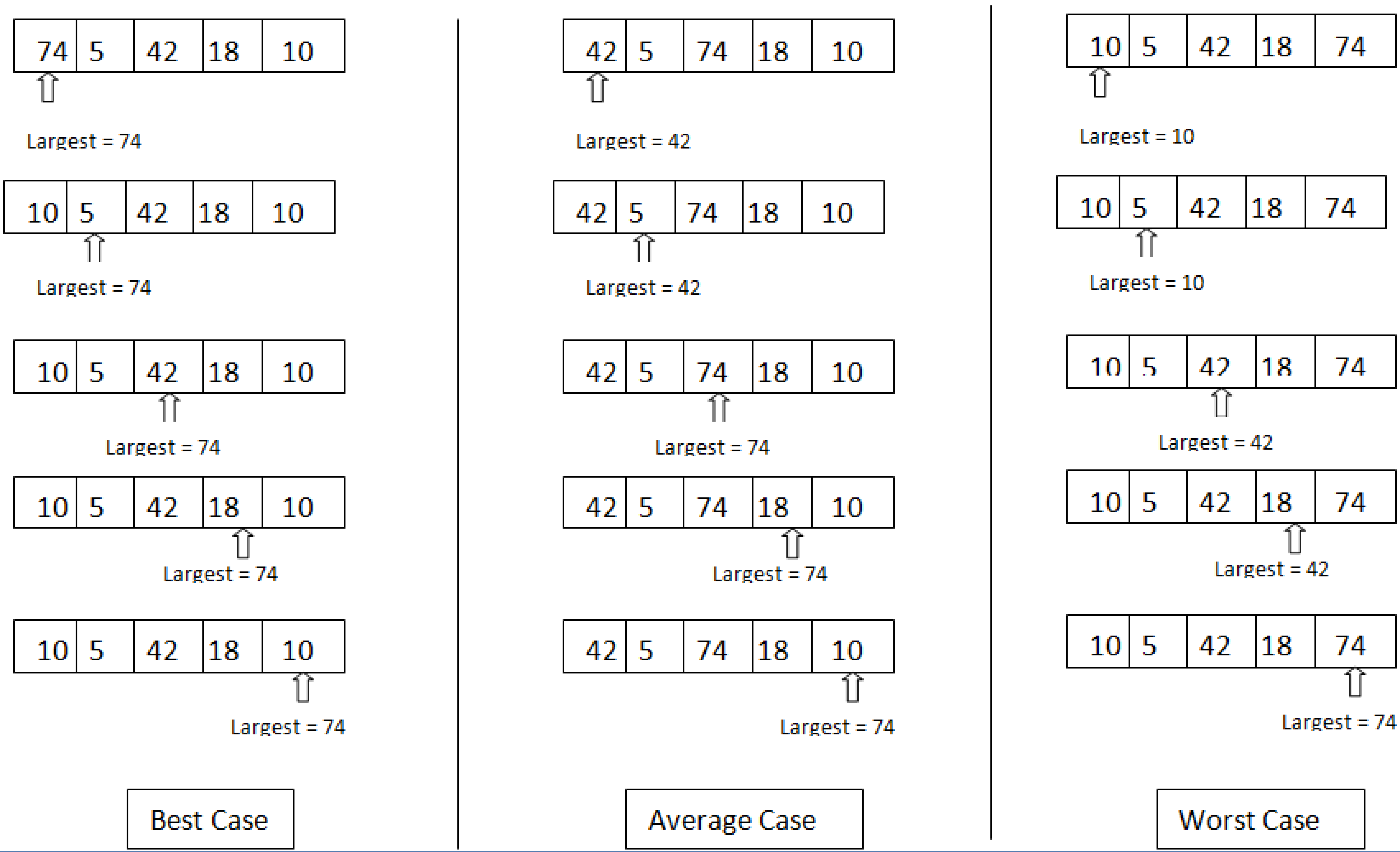
Time Complexity

- Core Factors
  - machine speed, language, compiler etc
- Generic Factors
  - Size of input
  - Basic operations performed
  - Class of input (best, average and worst)

Case 1

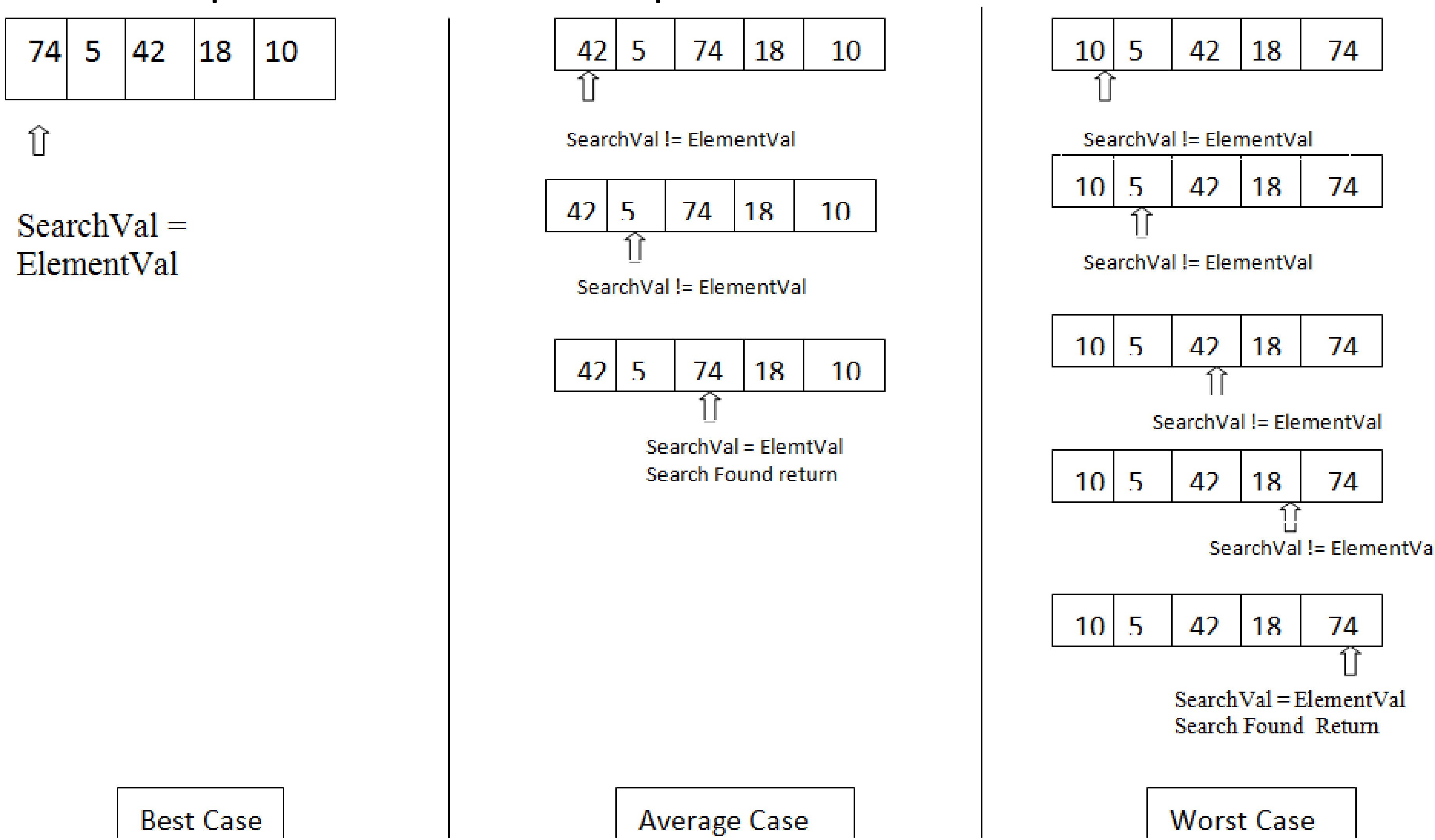
Dependent on size but independent of class of input

Find Largest Element in List



Case 2

Dependent on class of input as well as size



How to Calculate ?

Algorithm findLargest (list, n)

Finds the largest element in a list

Pre : list is an array which contains elements

n is the number of elements in list

Return : Value of largest element in list

```
set walker to 0                              // c1
set largest to -1                            // c2
loop( walker < n )                           // c3
    if( list[walker] > largest )             //  c4
        largest = list[walker]               //c5
    increment walker                         //c6
end loop
return largest                               //c7
```

Basic operations	Constant Symbol Purpose	Assumption Time in microsecond
c1	Time required to set memory location identified by variable walker to value 0	2
c2	Time required to set memory location identified by variable largest to value -1	2
c3	Time required to compare values at location walker and n	5
c4	Time required to compare values residing at memory location largest and list[walker]	5
c5	Time required set value at location largest with that of location list[walker]	2
c6	Time required increment the value residing at location walker	1
c7	Returning value at location identified by variable largest	1

$$T(n) = c1 + c2 + [\ \sum_{walker=0}^{n-1}(c3 + c4 + c5 + c6)\ ] + c7$$

$$T(n) = c1 + c2 + c7 + (c3+c4+c5+c6) \sum_{walker=0}^{n-1} 1$$

$$T(n) = c1 + c2 + c7 + n * (c3+c4+c5+c6)$$ **Formula -** $c \sum_{i=0}^{n} 1 = c * (n+1)$

Size huge Factor

Size – n	T(n) = c1 + c2 + c7 + n *(c3+c4+c5+c6)	Time in microseconds
10	T(10) = 2 + 2 + 1 + 10 * (5 + 5 + 2 + 1)	135
100	T(100) = 2 + 2 + 1 + 100 * (5 + 5 + 2 + 1)	1305
1000	T(1000) = 2 + 2 + 1 + 1000 * (5 + 5 + 2 + 1)	13005
10000	T(10000) = 2 + 2 + 1 + 10000 * (5 + 5 + 2 + 1)	130005

Asymptotic Analysis

- Focus on the dominant factor
  - The number of times the basic operations are repeated.
  - The two dominant factors are loops and recursion
- Asymptotic Notations
  - Big O : Upper Bound
  - Big Omega : Lower bound
  - Theta

Analyze Loops

- Linear Loop
  - A loop has linear factor when increment | decrement consist of addition or subtraction of constant c

Addition

```
for(i=0 ; i< n ; i= i+1
    c=a+b
```

Subtraction

```
for(i=n ; i>0 ; i= i-1 )
    c=a+b
```

- Thus the execution is linearly dependent on value of n
  $T(n) = n$

n	No of time c is executed
10	10
100	100
1000	1000
10000	10000

- Logarithmic Loop :
  - The controlling variable is divided or multiplied in each iteration

Multiply

```
for(i=1; i< n ; i= i*2 )
    c=a+b
```

Divide Loop

```
for(i=n ; i>0 ; i= i/2 )
    c=a+b
```

Multiply		Divide	
Iteration	Value of i	Iteration	Value of i
0	1	0	100
1	2	1	50
2	4	2	25
3	8	3	12
4	16	4	6
5	32	5	3
6	64	6	1

$2^{iteration} <= n$

Taking Log

$\text{Log}_2\, n = \text{iteration}$

$n / 2^{iteration} >= 1$

Taking Log

$\text{Log}_2\, n = \text{iteration}$

- Nested Loops
  - Loops that contain Loops are called nested loops
  - Total_iterations = outer_loop_iteration * inner_loop_iterations

i. Linear Logarithmic : The outer loop executes linear complexity and inner loop has logarithmic complexity

```
for( i = 1 ; i <= n ; i++ )          // n times
  for(j = 1 ; j <= n ; j= j*2 )          // log n times
    //code
```

T(n) = n * log n

ii. Quadratic : The inner & outer loop both execute n times

```
for( i = 0 ; i < n ; i++ )              // n times
        for( j = 0 ; j < n ; j++ )      //  n times
                code
T(n) = n * n
= n²
```

iii. Dependent Quadratic : The inner loop depends upon outer loop

```
for( i = 0 ; i < n ; i++ )
        for( j  = 0 ; j <= i ; j++ )
                // code
```

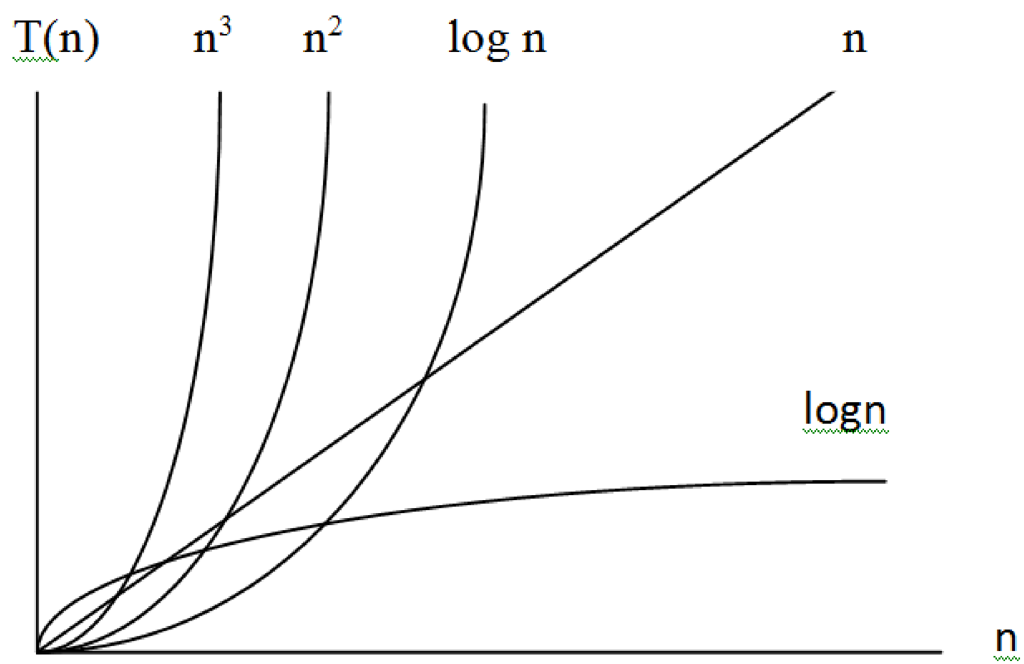
Outer Iteration No	No of times inner loop executes
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

The equation of quadratic loop is given by outer_loop_iteration * average_of_inner loop_iteration

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

Dominant Factor

Efficiency	T(n) for n = 10000	Iterations	Estimated Time
Logarithmic	log n	14	Microseconds
Linear	N	10000	Seconds
Linear logarithmic	Nlogn	140000	Seconds
Quadratic	N^2	10000^2	Minutes
Polynomial	N^k	10000^k	Hours
Exponential	C^n	2^{10000}	Intractable
Factorial	N!	10000!	Intractable



Contact Info

- trainers@finaldesk.com
- rishabh@finaldesk.com
- nilesh@finaldesk.com
- jignesh@finaldesk.com
- yash@finaldesk.com
- anand@finaldesk.com