Asymptotic Notations and Time Complexity

Dr. Viswanathan V.
Professor
School of Computer Science and Engineering
Vellore Institute of Technology, Chennai

Algorithm analysis

- Algorithm - Step by step procedure to solve the problem/task

- Solving a problem - Different method/procedure

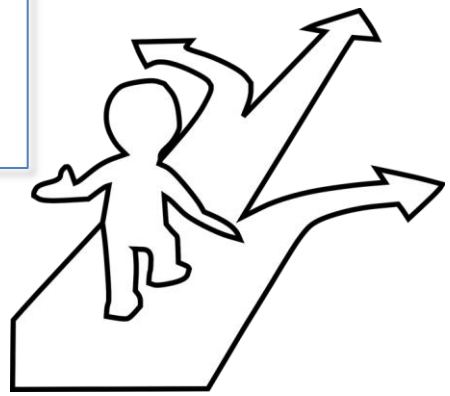
- **Problem:** Sum of first 'n' natural numbers

Solution 1:

input n
sum=0
for i = 1 to n
sum = sum + i
print sum

Solution 2:

input n
*sum = n * (n+1) / 2*
print sum



- Choosing the best method

How do we compare algorithms?

We need to define a number of objective measures.

1. Compare execution times?

   - times are specific to a particular computer - ***Not good***

2. Count the number of statements executed?

   - number of statements vary with the programming language as well as the style of the individual programmer. - ***Not good***

How can I measure the performance of an algorithm?

Challenge : Finding the most efficient algorithm for solving a problem.

- Space Complexity
  - How much space is required
    - The amount of memory required by an algorithm to completion
    - Some algorithms may be more efficient if data completely loaded into memory

- Time complexity
  - How much time does it take to run the algorithm.
  - It should be independent of machine time, programming style, etc.

Measuring the performance of an algorithm ...

Space complexity of an algorithm

The total amount of the computer's memory used by an algorithm when it is executed is the space complexity of that algorithm.

Time complexity of an algorithm

The time complexity is the number of operations an algorithm performs to complete its task regardless kind of machine it runs on.

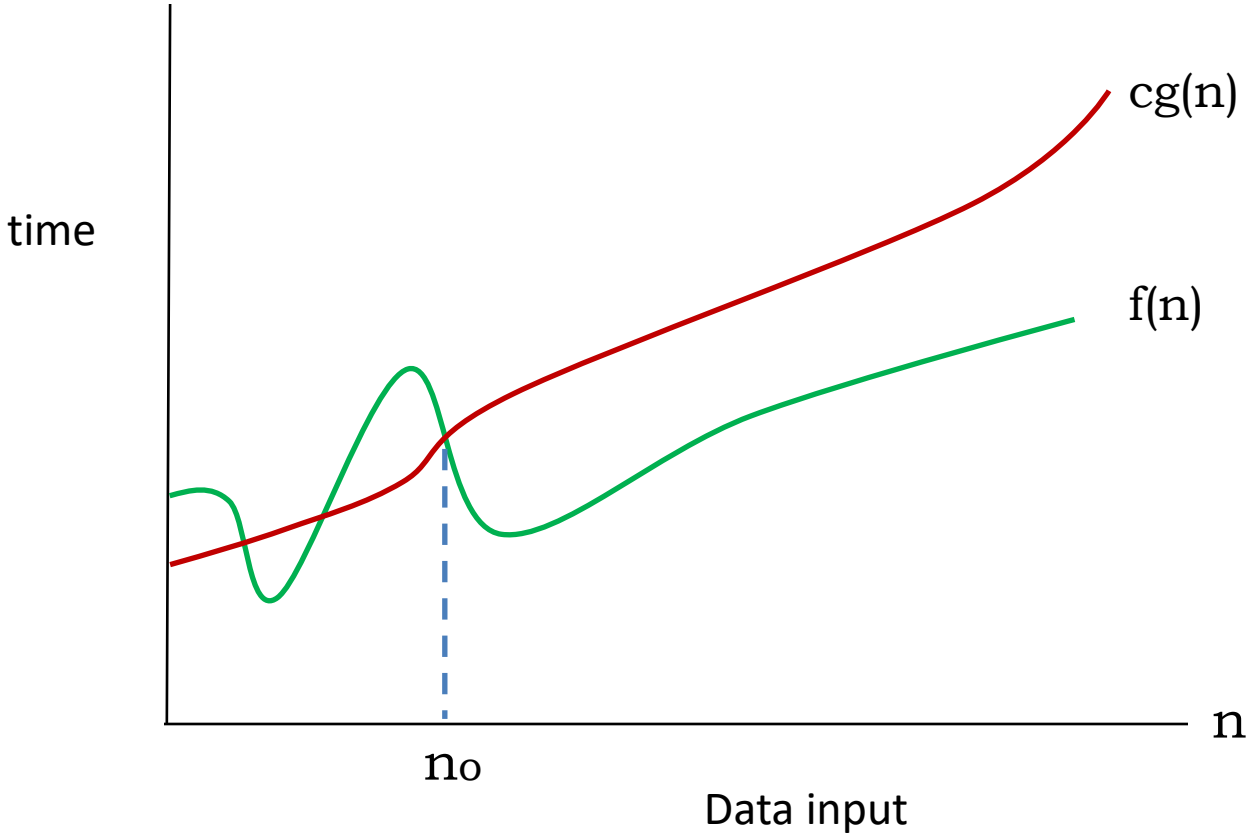
Asymptotic notations

Asymptotic Analysis is to measure the performance of an algorithm in terms of **input size** (without measuring the actual running time) and calculate, how the time (or space) taken by an algorithm increases with the input size. This is also known as an algorithm's growth rate.

Three types of **asymptotic notations** to represent the growth of any algorithm

- Big Oh(O)
- Big Omega (Ω)
- Big Theta (Θ)

Big O notation



$f(n) \leq c\, g(n)$, $c>0$ *and* $n_o \geq 1$, $n \geq n_o$

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

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

Example :

Big 'O'

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

$f(n) \leq c\, g(n)$, $c \geq 4$ *and* $n_o \geq 2$

$f(n) = O(n)$

Proof:

$3n+2 \leq c\, n$

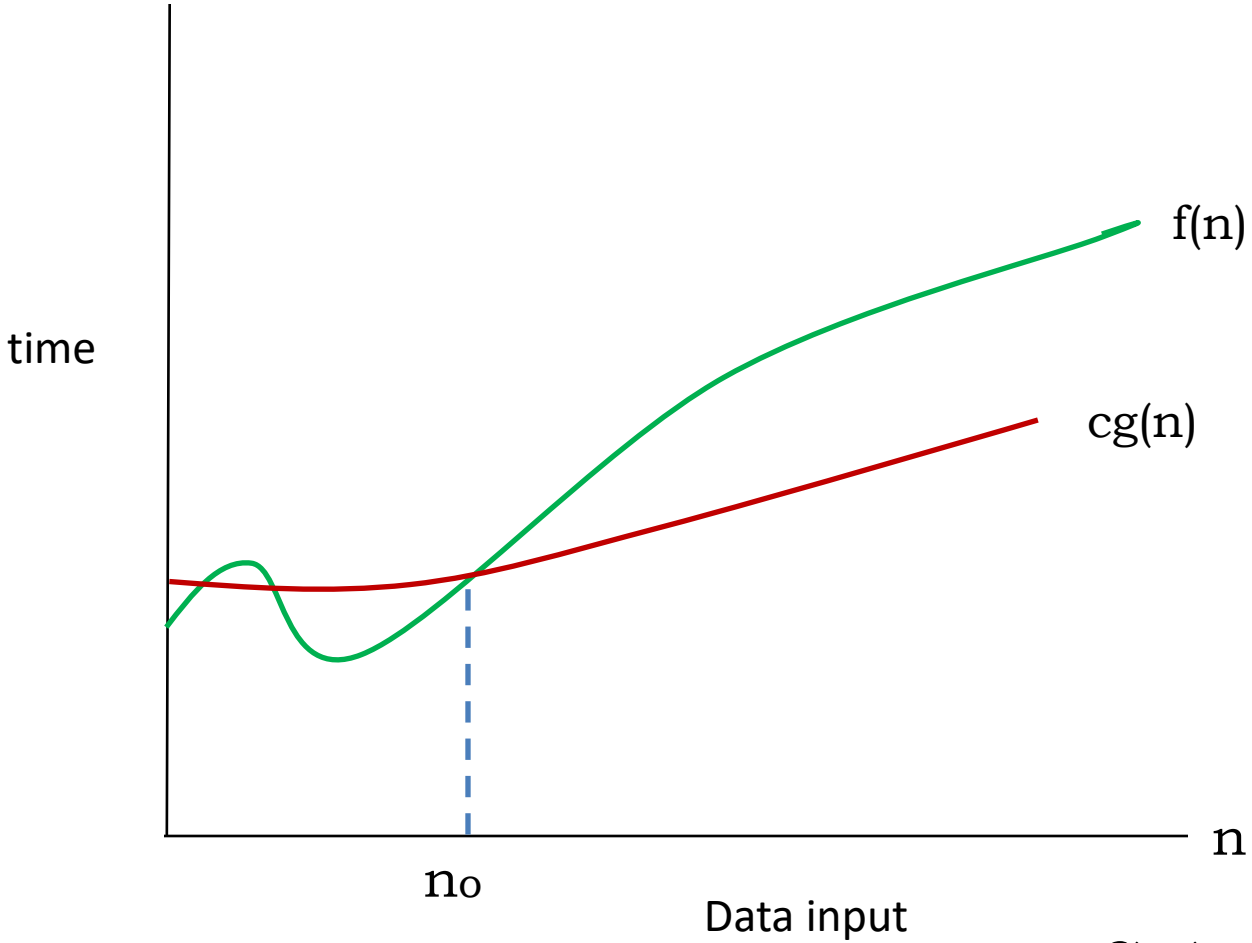
When c=4; $3n+2 \leq 4\, n$

$n \geq 2$

Hence when $n_o \geq 2$ *and* $c \geq 4$ $f(n) \leq c\, g(n)$

$\therefore \quad f(n) = O(n)$

Ω - notation



f(n) = Ω(g(n))

$f(n) \geq c\, g(n)$, $c>0$ *and* $n_o \geq n$

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

Example :

Big 'Ω'

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

$f(n) \geq c\, g(n)$, $c>1$ *and* $n_o \geq 1$

$f(n) = \Omega(n)$

Proof:

$3n+2 \geq c\, n$

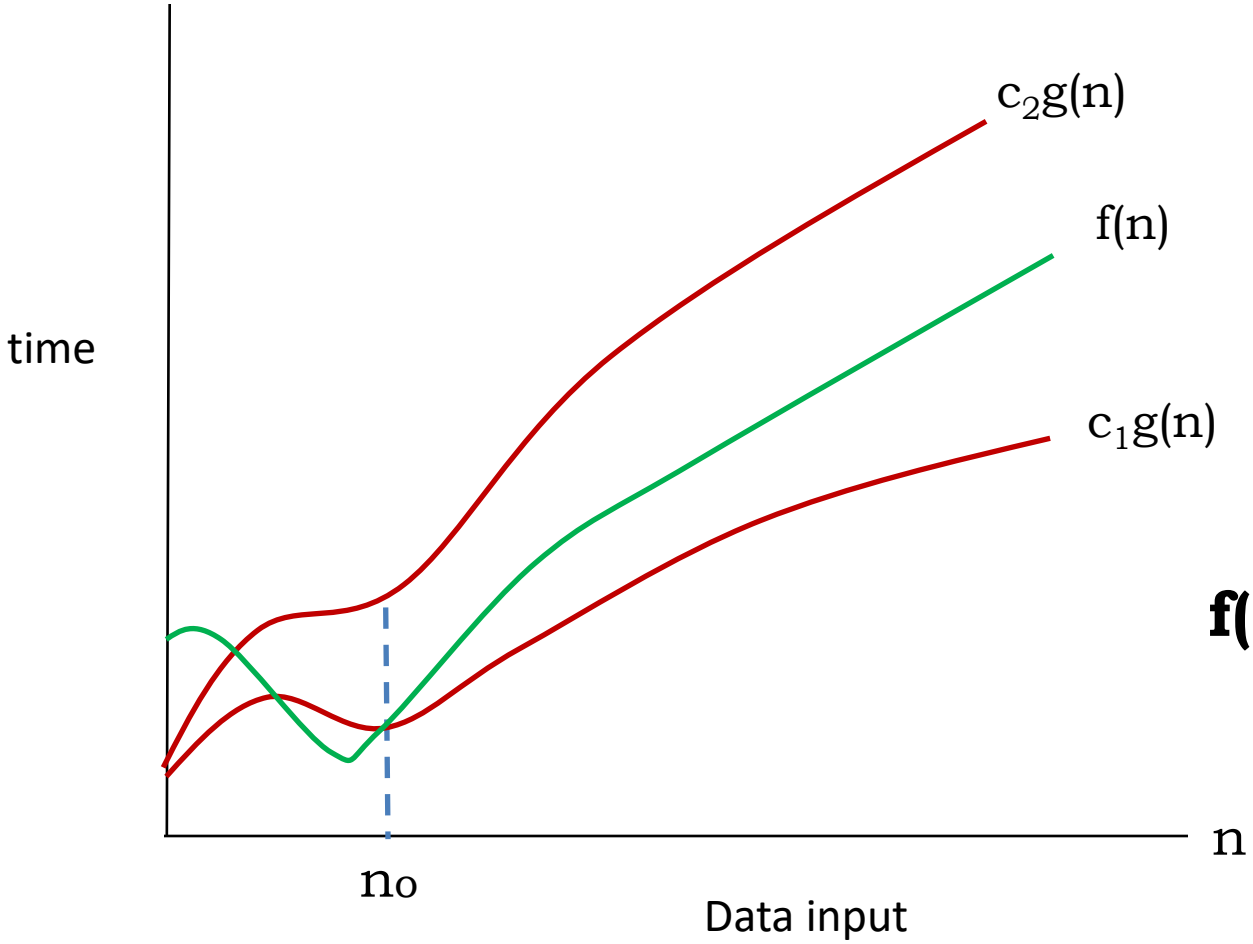
When c=1; $3n+2 \geq n$

$n \geq 1$

Hence when $n_o \geq 1$ *and* $c = 1$ $f(n) \geq c\, g(n)$

$\therefore$ $f(n) = \Omega(n)$

Θ - notation



f(n) = Θ(g(n))

$c_1g(n) \leq f(n) \leq c_2g(n)$ $\quad c_1, c_2 > 0$ *and* $n_o \geq 1$, $n \geq n_o$

$f(n) = \Theta(g(n))$

Example :

Big 'Θ'

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

$f(n) \leq c_2 g(n)$, $c_2 \geq 4$ *and* $n_o \geq 1$

$f(n) \geq c_1 g(n)$, $c_1 \geq 1$ *and* $n_o \geq 1$

$f(n) = \Omega(n)$

Proof:

$3n+2 \geq c\,n$

When c=1; $3n+2 \geq n$

$n \geq 1$

When c=4; $3n+2 \leq 4\,n$

$n \geq 1$

Hence when $n_o \geq 1$ *and* $c \geq 1$ $\quad c_1\, g(n) \leq f(n) \leq c_2 g(n)$

$\therefore \quad f(n) = \Theta(n)$

Common time complexities

Constant time - O(1)

Ex: Odd or Even number

O(1) describes algorithms that take the same amount of time to compute regardless of the input size.

For instance, if a function takes the identical time to process ten elements as well as 1 million items, then we say that it has a constant growth rate or O(1).

Logarithmic time - O(log n)

Ex: Finding element on sorted array with binary search

Logarithmic time complexities usually apply to algorithms that divide problems in half every time.

Common time complexities ....

Linear time - O(n)

Ex: Find max element in unsorted array,
Linear time complexity O(n) means that as the input grows, the algorithms take proportionally longer to complete.

Line arithmic time - O(n log n)

Ex: Sorting elements in array with merge sort

Common time complexities ...

Quadratic time - $O(n^2)$

Ex: Duplicate elements in array

A function with a quadratic time complexity has a growth rate of n^2. If the input is size 2, it will do four operations. If the input is size 8, it will take 64, and so on.

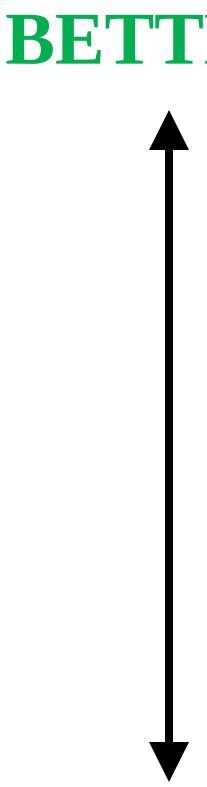
Polynomial time - $O(n^c)$ $c>1$

Polynomial running is represented as $O(n^c)$, when $c > 1$.

Exponential time - $O(2^n)$

Ex: Find all subsets

Common time complexities ....



$O(1)$	constant time
$O(\log n)$	log time
$O(n)$	linear time
$O(n \log n)$	log linear time
$O(n^2)$	quadratic time
$O(n^3)$	cubic time
$O(2^n)$	exponential time

Types of Analysis

Worst case (*O* notation)

- we calculate upper bound on running time on algorithm.
- An absolute guarantee that the algorithm would not run longer
- No matter what the inputs are. (Maximum number of operations to be executed to complete the task)

Average case (Θ notation)

In the average case, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs.

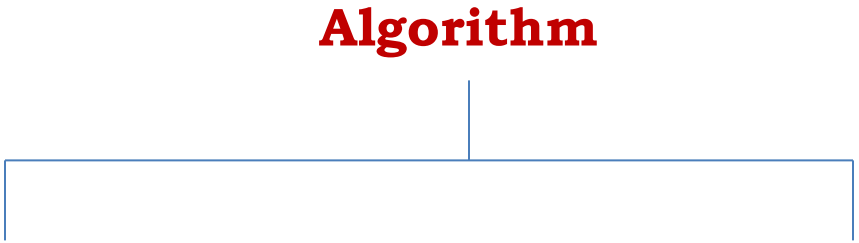
Types of Analysis ….

Best case (Ω notation)

In the best case analysis, we calculate lower bound on running time on algorithm. Input is the one for which the algorithm runs the fastest.

ie. Minimum number of operations to be executed to complete the task.

Generally We will find **worst case time complexity**



Iterative Algorithm

```
Algorithm X()
{
 ……
for i= 1 to n
  x=x+1
……
}
```

Recursive Algorithm

```
Algorithm X(a)
{ …….

  if (a<10)
      X(a/2)
   ……..
}
```

Iterative program - Time complexity is calculated by the number of time the loop will execute

Recursive program - Time complexity is calculated by the recursive equation. We will write the f(a) in terms of f(a/2)

If there is no iteration or recursion in the program, then the time complexity of an algorithm is constant time. ie. $O(1)$

Time complexity of Iterative algorithms

General Plan for Analysis

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules

Example 1:

```
Algorithm X()
{

for i = 1 to n
   print("VIT")
}
```

basic operations

Number of time the basic operations will execute : n

Time complexity of an algorithm is $O(n)$

Example 2:

```
Algorithm X()
{

for i = 1 to n
      for j = 1 to n
         print("VIT")
}
```

basic operations

Number of time the basic operations will execute : n^2

Time complexity of an algorithm is $O(n^2)$

Example 3:

```
Algorithm X()
{
    while(j<=n)
     {
       i = i+1
       j = j + i
       print("VIT")
     }
}
```

basic operations

i	1	2	3	4	5	6	…	k
j	1	3	6	10	15	21	⋯	n

2*(2+1)/2

4*(4+1)/2

$$\frac{k\,(k+1)}{2} > n$$

$$\frac{(k^2+k)}{2} > n$$

$$k = O(\sqrt{n})$$

Example 4:

```
Algorithm X()
{
   for i = 1 to n
   {
      for j = 1 to i
       {
          for k= 1 to 100
              print("VIT")
       }
   }
}
```

i	1	2	3	4	….	n
j	1	2	3	4	….	n
k	100	200	300	400	….	n x 100

100+200+300+…….+n*100

100(1+2+3+…+n)

$$100 * \frac{(n^2 + n)}{2}$$

Time complexity of an algorithm is $O(n^2)$

Example 5:

```
Algorithm X()
{

   while(i<n)
  {
        print("VIT")
        i = i *2

}
```

i =	1	2	4	8	.....	n
	2^0	2^1	2^2	2^3		2^k

$$\boldsymbol{n = 2^k}$$

$$\boldsymbol{\log(n) = k}$$

Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic operation is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic operation is executed.
- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Methods for analyzing recursive algorithm

- Substitution method
- Master Theorem
- Recursion - tree method

Example : Recursive evaluation of *n*!

Definition: $n! = 1 * 2 * \ldots * (n-1) * n$ for $n \geq 1$ and $0! = 1$

Recursive definition of *n*!: $F(n) = F(n-1) * n$ for $n \geq 1$ and $F(0) = 1$

Algorithm F(n)
//Computes n! recursively
//Input : A non negative integer 'n'
//Output : The value of n!
if n=0 return 1
else return ***F(n-1)*** * *n*

M - multiplications

Size: n
Basic operation: multiplication
Recurrence relation: $M(n) = M(n-1) + 1$
$M(0) = 0$

Solving the recurrence for M(*n*)

$M(n) = M(n-1) + 1, \quad M(0) = 0$

Backward substitution.

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

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

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

…

$= M(n-i) + i$

$= M(0) + n$

$= n$

Hence the time complexity is ***O(n)***

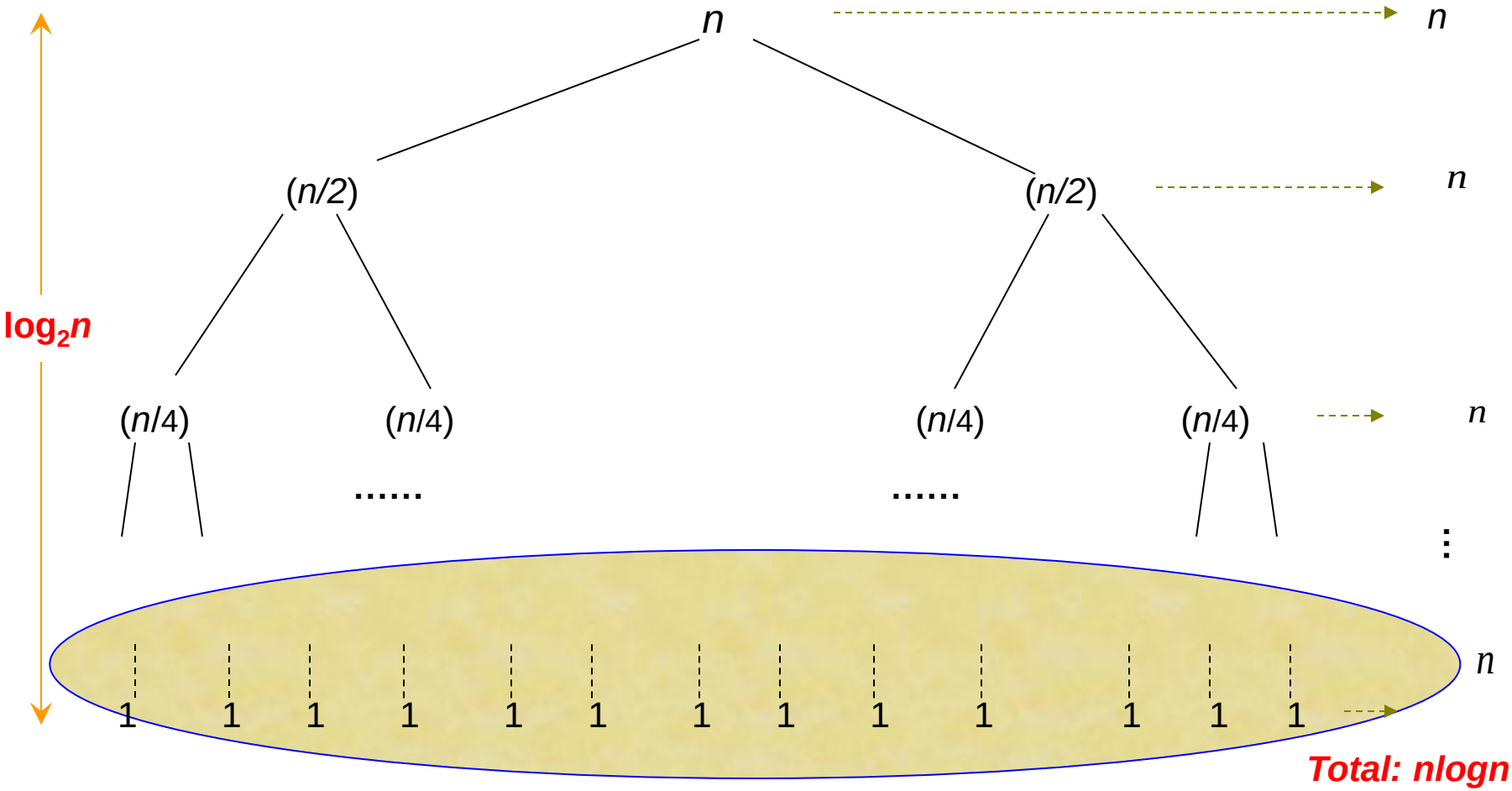
Guess and verify method

$M(0) = 0$
$M(1) = M(1-1)+1 = M(0)+1 = 1$
$M(2) = M(2-1)+1 = M(1)+1 = 2$
$M(3) = M(3-1)+1 = 2+1 = 3$

Therefore $M(n) = n$

Solve the recurrence relation using Recursion Tree

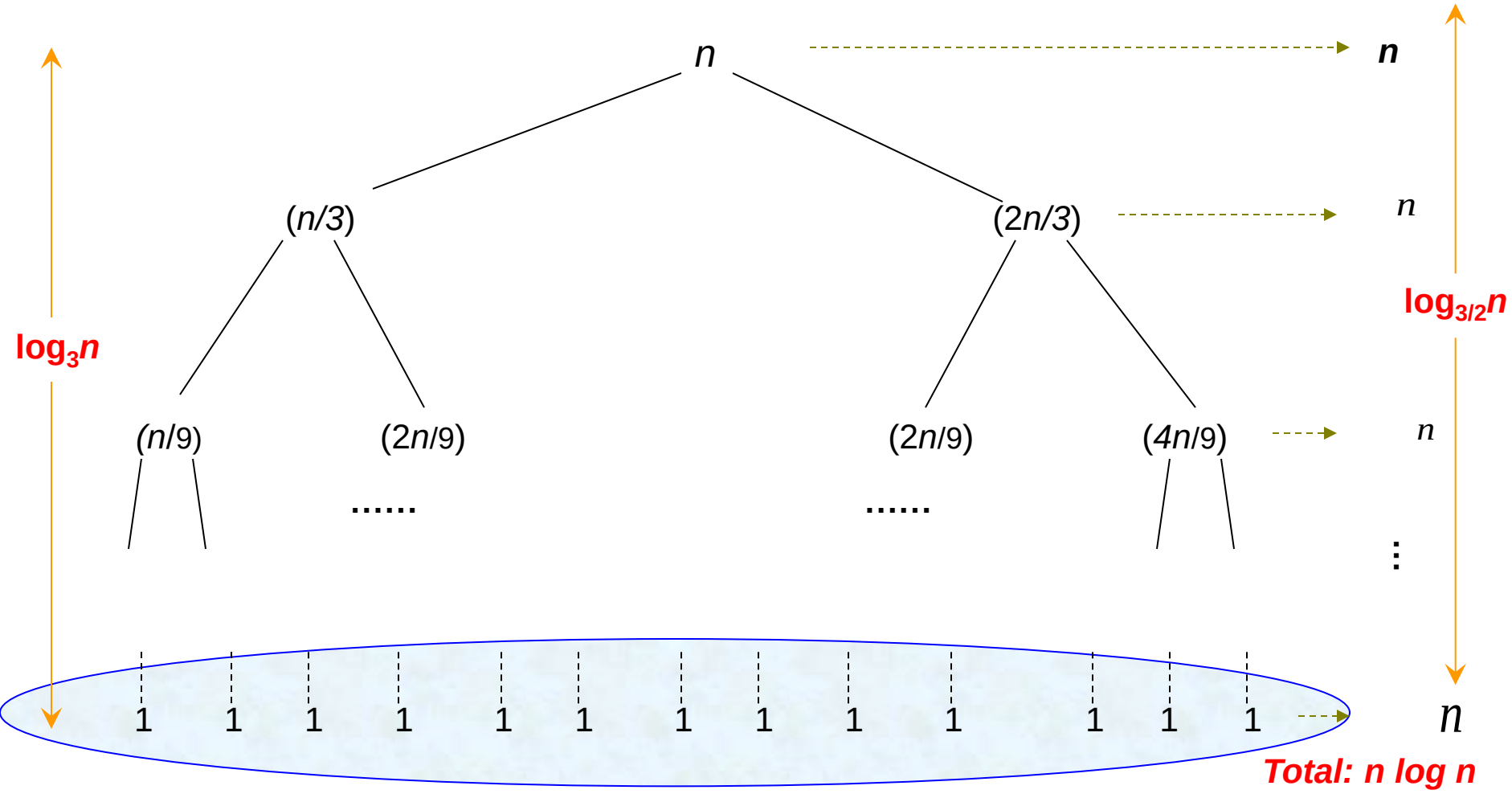
Example 1: T(n) = 2T(n/2)+n



Time complexity is O (n log n)

Solve the recurrence relation using Recursion Tree …

Example 2: T(n) = T(n/3)+T(2n/3) +n



Time complexity is O**(nlogn)**

Master Theorem

Master theorem is used for calculate the time complexity of recurrence relations (divide and conquer algorithms) in a simple and quick way.

$T(n) = aT(n/b) + f(n)$, where $f(n) \in \Theta(n^k)$, $k>=0$

where,

n = size of input

a = number of sub problems in the recursion

n/b = size of each sub problem.

f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

Here, a ≥ 1 and b > 1 are constants,

Master Theorem ....

Cases :

1. $a < b^k$ $\quad T(n) \in \Theta(n^k)$
2. $a = b^k$ $\quad T(n) \in \Theta(n^k \log n)$
3. $a > b^k$ $\quad T(n) \in \Theta(n^{\log_b a})$

- Examples:
  - $T(n) = T(n/2) + 1$ $\quad \Theta(\log n)$
  - $T(n) = 2T(n/2) + n$ $\quad \Theta(n \log n)$
  - $T(n) = 3T(n/2) + n$ $\quad \Theta(n^{\log_2 3})$
  - $T(n) = T(n/2) + n$ $\quad \Theta(n)$

Thank you