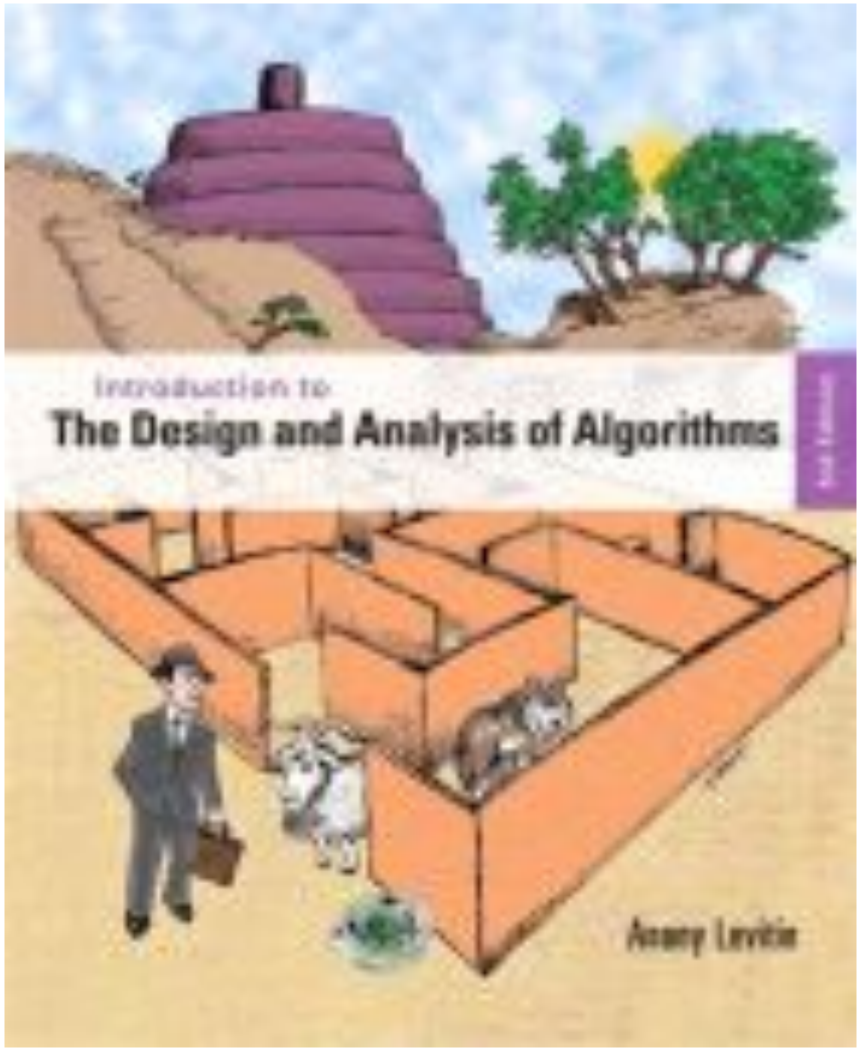
Analysis and Design of Algorithms
(19CS4PCADA)

Google class code: iar23so

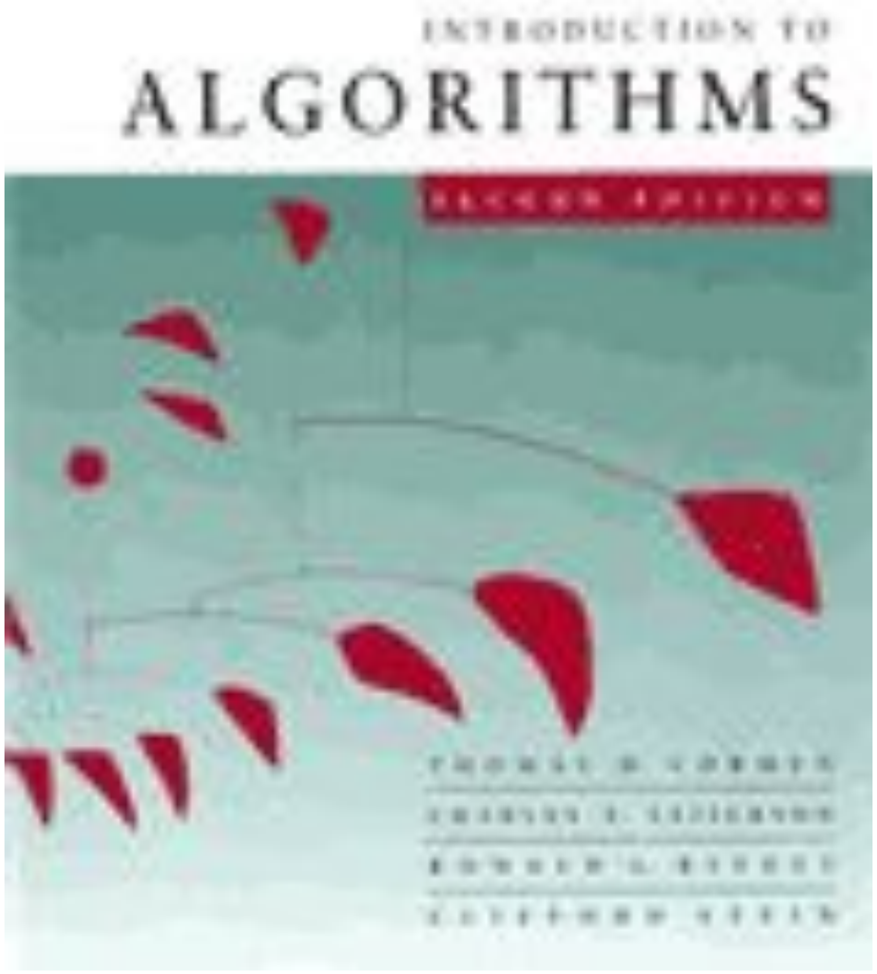
Textbook

- *Anany Levitin*
- **Introduction to The Design & Analysis of Algorithms**.
- Addison-Wesley



Textbook

- T. **C**ormen, C. **L**eiserson, R. **R**ivest, and C. Stein
- Introduction to Algorithms, 3rd Ed.
- MIT Press and McGraw-Hill Book Company



Unit 1: Introduction to Algorithms

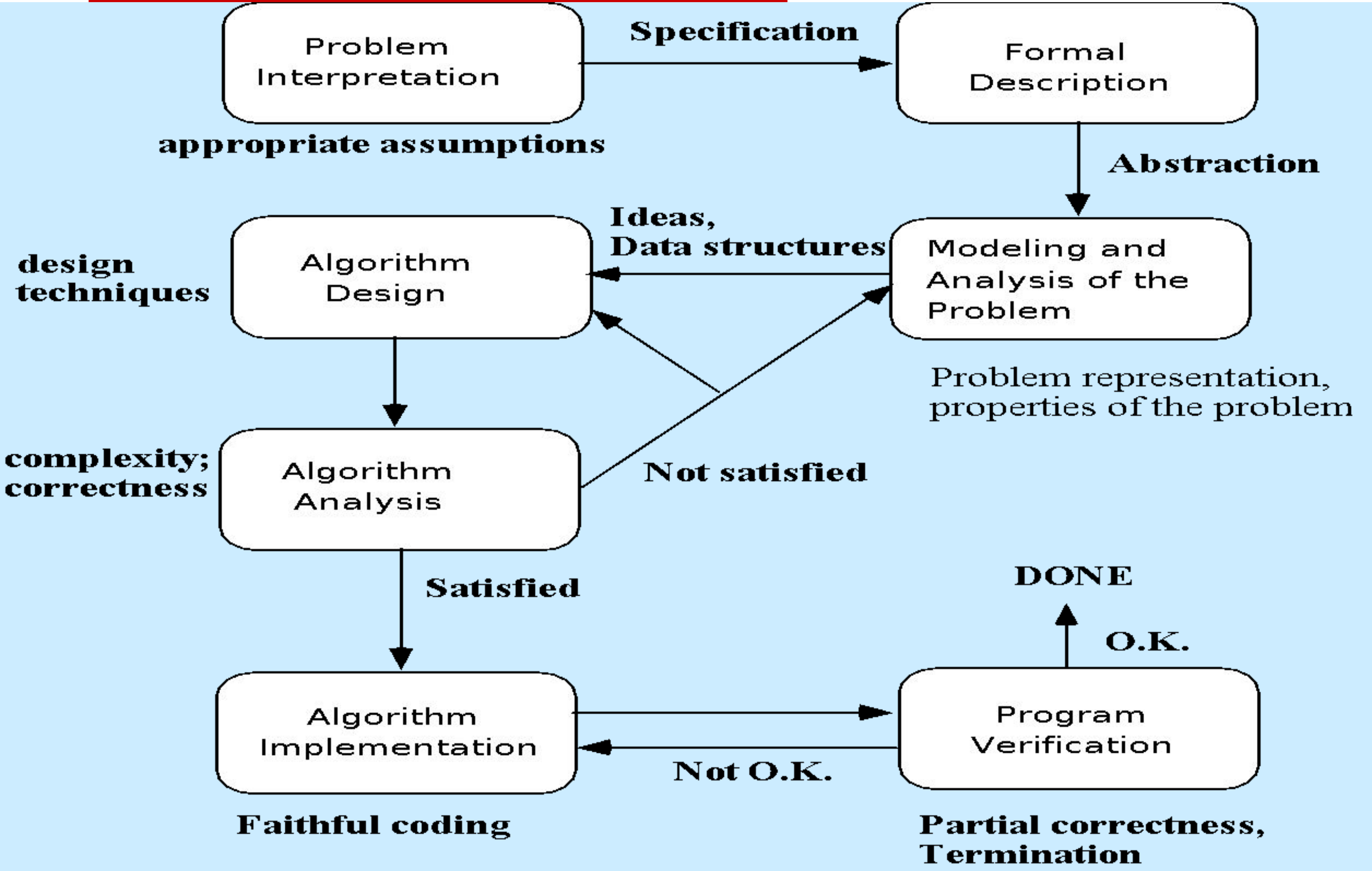
- Fundamentals of Algorithmic Problem Solving
- Space and Time Complexity
- Order of Growth
- Asymptotic Notations

Computer Science Algorithms

- For a Computer Program to accomplish the Task

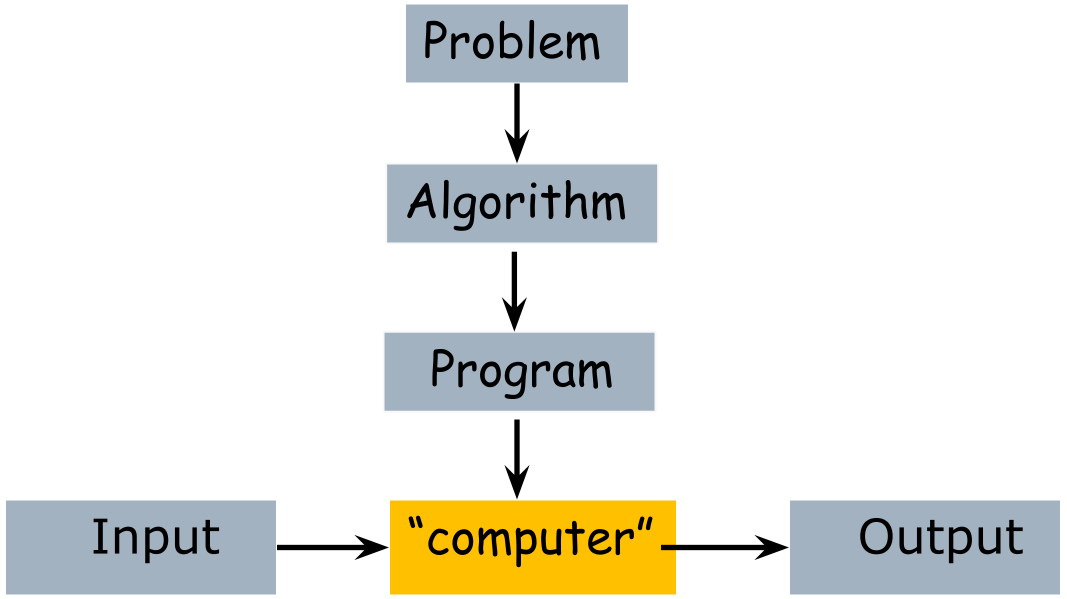
$$\begin{tikzcd}
\text{Start with Input Data} \arrow[d] \\
\text{Do Some Complex Calculations} \arrow[d] \\
\text{Stop when we find Answer}
\end{tikzcd}$$

Life Cycle of Software Development



Algorithm

- An algorithm is a sequence of unambiguous instructions for solving a computational problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



Examples of Algorithms

Computing **Greatest Common Divisor** of Two non-negative, not-both zero Integers

- gcd(m, n): the largest integer that divides both *m* and *n*
- First try - Euclid's Algorithm:
  - Idea: gcd(m, n) = gcd(n, m mod n)

Greatest Common Divisor (Euclid’s Algorithm), **gcd(m, n)**

- **Step 1**: If n = 0, return value of m as the answer and stop; otherwise, proceed to Step 2.
- **Step 2**: Divide m by n and assign the value of the remainder to r.
- **Step 3:** Assign the value of n to m and the value of r to n. Go to Step 1.

Pseudocode for (Euclid’s Algorithm), **gcd(m, n)**

```
ALGORITHM Euclid(m, n)
// Computes gcd(m, n) by Euclid’s algorithm
// Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while n ≠ 0 do
    r = m mod n
    m = n
    n = r
return m
```

r=36 m=48 n=36

r=12 m=36 n=12

r=0 **m=12** n=0

Question:
GCD(36,48) how many division Operations are required to compute GCD using Euclid algorithm ?

Second try: Middle-school procedure, **gcd(m, n)**

- **Step 1**: Find prime factors of m.
- **Step 2**: Find prime factors of n.
- **Step 3**: Identify all common prime factors of m and n
- **Step 4**: Compute product of all common factors and return product as the answer.

PF of 36: 1,2,2,3,3

PF of 48: 1,2,2,2,2,3

CF: 1*2*2*3=12

Question:
GCD(36,48) how many division Operations are required to compute GCD using Middle-School procedure ?

Third try: Consecutive Integer Checking, **gcd(m, n)**

- **Step 1:** Assign the value of min{m, n} to q.
- **Step 2:** Divide m by q. If the remainder is 0, go to Step 3; otherwise, go to Step 4.
- **Step 3:** Divide n by q. If the remainder is 0, return the value of q as the answer and stop; otherwise, proceed to Step 4.
- **Step 4:** Decrease the value of q by 1. Go to Step 2.
- q=36

Question:
Try computing GCD(36,48) using Consecutive Integer Checking method ?

What can we learn from the three examples of gcd(m, n) ?

- Each step must be basic and unambiguous
- Same algorithm, but different representations (different pseudocodes)
- Same problem, but different algorithms, based on different ideas and having dramatically different speeds.
  - gcd(31415, 14142) = 1; Euclid takes ~0.08 ms whereas Consecutive Integer Checking takes ~0.55 ms, about 7 times speedier

Method 1: Greatest Common Divisor (Euclid's Algorithm), **gcd(m, n)**

- **Step 1**: If n = 0, return value of m as the answer and stop; otherwise, proceed to Step 2.
- **Step 2**: Divide m by n and assign the value of the remainder to r.
- **Step 3:** Assign the value of n to m and the value of r to n. Go to Step 1.

Method 2: Consecutive Integer Checking, **gcd(m, n)**

- **Step 1:** Assign the value of min{m, n} to q.
- **Step 2:** Divide m by q. If the remainder is 0, go to Step 3; otherwise, go to Step 4.
- **Step 3:** Divide n by q. If the remainder is 0, return the value of q as the answer and stop; otherwise, proceed to Step 4.
- **Step 4:** Decrease the value of q by 1. Go to Step 2.

C++ Program - Analysis of the methods to find the GCD of two numbers

```
#include<iostream.h>
#include<conio.h>
#include<time.h>

long int euclid(long int m,long int n)
{
        clock_t start,end;
        start=clock();
        long int r;
        while(n!=0)
        {
                r=m%n;
                m=n;
                n=r;
        }
        end=clock();
        cout<<endl<<"Time taken:"<<(end-start)/CLK_TCK<<" sec";
        return m;
}
```

C++ Program - Analysis of the methods to find the GCD of two numbers

```
long int con(long int m,long int n)
{
        clock_t start,end;
        start=clock();
        long int t,r,g;
        if(m>n)
        {  t=n; }
        else
        {   t=m; }

       a:do
        {
                r=m%t;
                if(r!=0)
                t--;
        } while(r!=0);
        if(r==0)
        {
                r=n%t;
                if(r==0)
                g=t;
                else
                {
                t--;
                goto a;
                }
        }
        end=clock();
        cout<<"Time taken
:"<<(end-start)/CLK_TCK<<" sec";
        return g;
} /*End of the function con*/
```

C++ Program - Analysis of the methods to find the GCD of two numbers

```
void main()
{
		long int x,y;
		clrscr();

		cout<<"\t\t\tANALYSIS OF THE TWO ALGORITHMS"<<endl<<endl;
		cout<<"GCD - EUCLID'S ALG : "<<endl;
		cout<<"enter two numbers:";
		cin>>x>>y;
		cout<<endl<<endl<<"GCD : "<<euclid(x,y);
		cout<<endl<<endl<<"----------------------------------------------";
		cout<<endl<<endl<<"GCD - CONSECUTIVE INTEGER CHECKING ALG :
		 "<<endl<<endl;
		cout<<endl<<endl<<"GCD : "<<con(x,y);
		getch();
}
```

ANALYSIS OF THE TWO LGORITHMS

```
GCD - EUCLID'S ALG :
enter two numbers:7896543     345678

Time taken: 0.08 millisecond
 GCD : 3

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

GCD - CONSECUTIVE INTEGER CHECKING ALG :

Time taken :0.55  millisecond
GCD : 3
```

INFERENCE:
The euclid’s method takes less time than the consecutive integer checking method and hence euclid’s method is better.

Iterative GCD

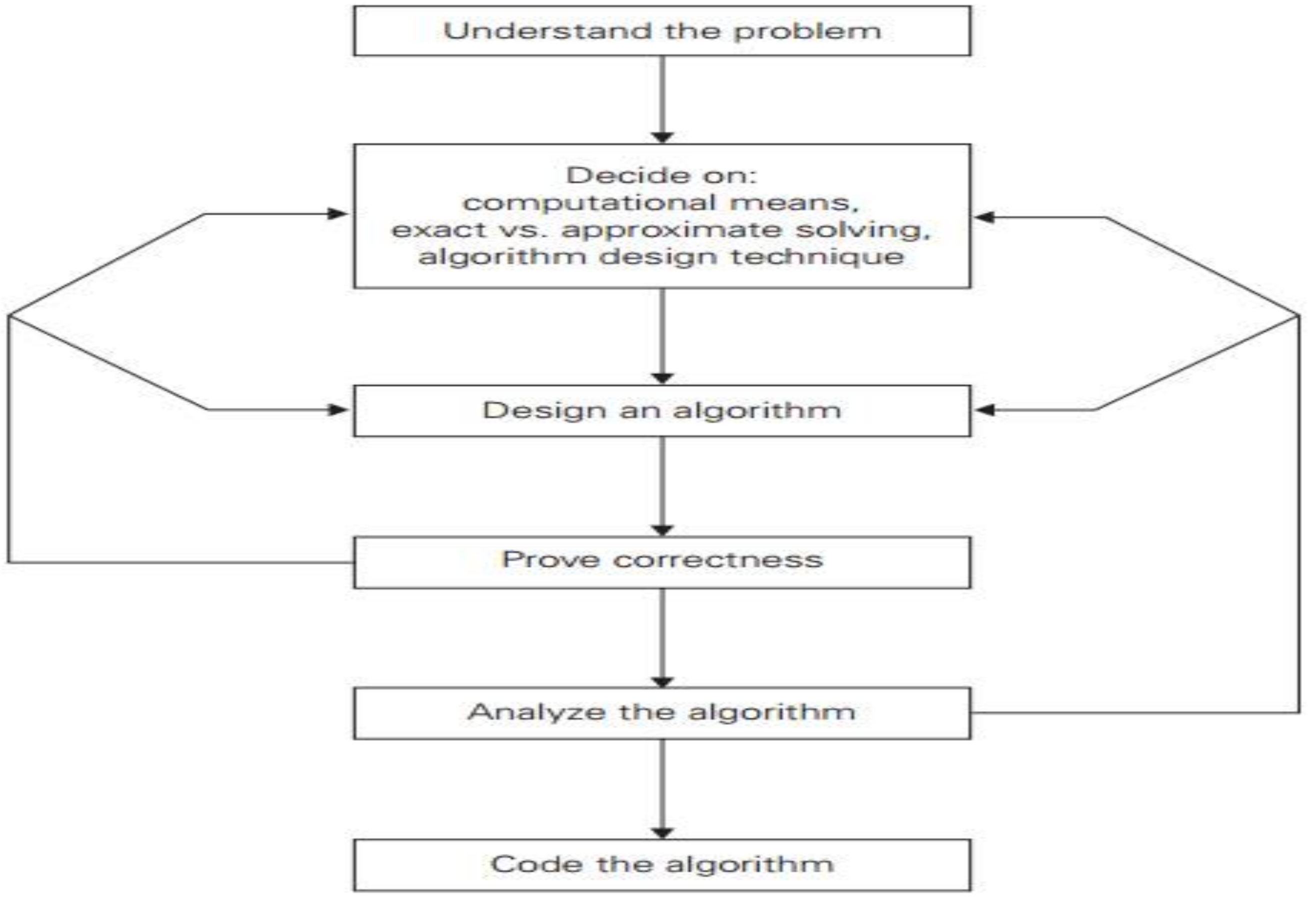
```
#include <stdio.h>
int main()
{
    int n1, n2, i, gcd;
    printf("Enter two integers: ");
    scanf("%d %d", &n1, &n2);

    for(i=1; i <= n1 && i <= n2; ++i)
    {
        // Checks if i is factor of both integers
        if(n1%i==0 && n2%i==0)
            gcd = i;
    }
    printf("G.C.D of %d and %d is %d", n1, n2, gcd);
    return 0;
}
```

Recursive GCD

```
#include <stdio.h>
int hcf(int n1, int n2);
int main() {
    int n1, n2;
    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);
    printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1, n2));
    return 0;
}

int hcf(int n1, int n2) {
    if (n2 != 0)
        return hcf(n2, n1 % n2);
    else
        return n1;
}
```

Fundamentals of Algorithmic Problem Solving

- Sequence of steps in the process of design and analysis of algorithms



Question

With the help of a flow chart, explain the various steps of algorithm design and analysis process.

Fundamentals of Algorithmic Problem Solving (Contd….)

- Understanding the problem
  - Ask questions, do a few small examples by hand, think about special cases, etc.
  - An input is an instance of the problem the algorithm solves
  - Specify exactly the set of instances the algorithm needs to handle
  - Example: gcd(m, n)

Fundamentals of Algorithmic Problem Solving (Contd….)

- Decide on
  - Exact vs. approximate solution
    - Approximate algorithm: Cannot solve exactly, e.g., extracting square roots, solving nonlinear equations, etc.
  - Appropriate Data Structure

Fundamentals of Algorithmic Problem Solving (Contd….)

- Design algorithm
- Prove correctness of the algorithm
  - Yields required output for every legitimate input in finite time
  - E.g., Euclid’s: gcd(m, n) = gcd(n, m mod n)
    - Second integer gets smaller on every iteration, because (m mod n) can be 0, 1, …, n-1 thus less than n
    - The algorithm terminates when the second integer is 0

Fundamentals of Algorithmic Problem Solving (Contd….)

- Analyze algorithm
  - **Time efficiency:** How fast it runs
  - **Space efficiency:** How much extra memory it uses
  - **Simplicity:** Easier to understand, usually contains fewer bugs, sometimes simpler is more efficient, but not always!
- **Generality:** i.e. the generality of the problem the algorithm solves and the set of inputs it accepts.

Fundamentals of Algorithmic Problem Solving (Contd….)

- Coding algorithm
  - Write in a programming language for a real machine
  - Standard tricks:
    - Compute loop invariant (which does not change value in the loop) outside loop
    - Replace expensive operation by cheap ones

Discussion: Algorithms in your Life

What algorithms do you use in every day life? Do you think you could write a program to make them more efficient?

What algorithms do you think are used by your favorite Games and Apps?

Have you ever made an algorithm for a program? What did it do? Was it correct and efficient?

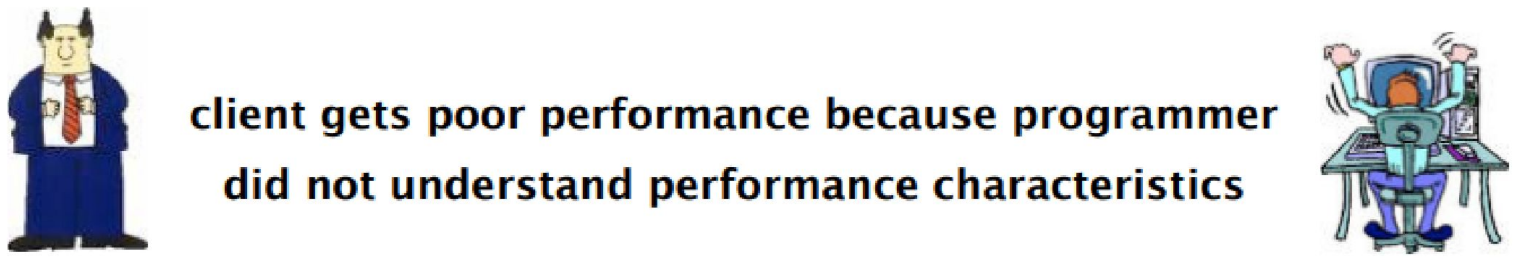
Analysis of Algorithms

Space Complexity
Time Complexity

Reasons to Analyze Algorithms

- Predict Performance
- Compare Algorithms
- Provide Guarantees
- Understand theoretical basis.

- Primary Practical Reason: Avoid Performance Bugs



client gets poor performance because programmer
did not understand performance characteristics

Performance measure of the algorithm

Two kinds of efficiency:
Space Efficiency or Space Complexity
Time Efficiency or Time Complexity

Two kinds of Algorithm Efficiency

- Analyzing the efficiency of an algorithm (or the complexity of an algorithm) means establishing the amount of computing resources needed to execute the algorithm. There are two types of resources:

• Memory space. It means the amount of space used to store all data processed by the algorithm.

• Running time. It means the time needed to execute all the operations specified in the algorithm.

Space efficiency: Deals with the space required by the algorithm

Time efficiency: It indicates how fast an algorithm runs.

What is Space complexity?

For any algorithm, memory is required for the following purposes...

- Memory required to store program instructions
- Memory required to store constant values
- Memory required to store variable values

Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm

What is Space complexity?

Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

- **Instruction Space:** ?
- **Data Space:** ?
- **Environmental Stack:** ?

What is Space complexity?

Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

- **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
- **Data Space:** It is the amount of memory used to store all the variables and constants.
- **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.

Space Complexity

Instruction space + Data space + Stack space

Calculating Space Complexity

To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following...

- 1 byte to store **Character** value,
- 2 bytes to store **Integer** value,
- 4 bytes to store **Floating** Point value,
- 6 or 8 bytes to store **double** value

Calculating Space Complexity

Example, Calculating the Data Space required for the following given code

```
int square(int a)
{
return a*a;
}
```

Calculating Space Complexity

Example, Calculating the Data Space required for the following given code

```
int square(int a)
{
return a*a;
}
```

	Data Space Required
For int a	2 Bytes
For returning a*a	2 Bytes
Total	**4 Bytes**

Calculating Space Complexity

Example:

```
int square(int a)
{
return a*a;
}
```

	Data Space Required
For int a	2 Bytes
For returning a*a	2 Bytes
Total	**4 Bytes**

Data Space Required:

- This code requires 2 bytes of memory to store variable '**a**' and another 2 bytes of memory is used for **return value**.
- That means, totally it requires **4 bytes of memory** to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be ***Constant Space Complexity***.
- If any **algorithm requires a fixed amount of space** for all input values then that space complexity is said to be **Constant Space Complexity**

Calculating Space Complexity

Example, Calculating the Data Space required for the following given code

```
int sum(int A[], int n) {
int sum = 0, i;
for(i = 0; i < n; i++)
sum = sum + A[i];
return sum;
}
```

Calculating Space Complexity

Example, Calculating the Data Space required for the following given code

```
int sum(int A[], int n) {
int sum = 0, i;
for(i = 0; i < n; i++)
sum = sum + A[i];
return sum;
}
```

	Data Space Required
For parameter int A[]	n *2 Bytes
For parameter n	2 Bytes
For local variable sum	2 Bytes
For local variable i	2 Bytes
Total	**2n+6 Bytes**

Calculating Space Complexity

```
int sum(int A[], int n) {
int sum = 0, i;
for(i = 0; i < n; i++)
sum = sum + A[i];
return sum; }
```

Data Space Required:

- '**n*2**' bytes of memory to store array variable '**a[]**'
  2 bytes of memory for integer parameter '**n**'
  4 bytes of memory for local integer variables '**sum**' and '**i**' (2 bytes each)
- That means, totally it requires **'2n+6' bytes of memory** to complete its execution. Here, the amount of memory depends on the input value of 'n'. This space complexity is said to be ***Linear Space Complexity*.**

If the **amount of space required by an algorithm is increased with the increase of input value**, then that space complexity is said to be **Linear Space Complexity**

Test your Knowledge

- Find Data Space required for the following code:

```
int sum(int x, int y, int z) {
 int r = x + y + z;
return r;
 }
```

Is the Space Complexity of this code is
"**Constant Space Complexity"**
or "**Linear Space Complexity" ?**

Test your Knowledge

- Find Data Space required for the following code:

```
void matrixAdd(int a[], int b[], int c[], int n) {
 for (int i = 0; i < n; ++i) {
 c[i] = a[i] + b[j]
 }
 }
```

Is the Space Complexity of this code is
"**Constant Space Complexity"**
or "**Linear Space Complexity" ?**

Performance measure of the algorithm

Two kinds of efficiency:

Space Efficiency or Space Complexity

Time Efficiency or Time Complexity

What is Time complexity?

- Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.

Time complexity of an algorithm can be defined as follows...

- **The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.**

What is Time complexity?

Time complexity of an algorithm can be defined as follows...

- **The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.**

Generally, running time of an algorithm depends upon the following...

- Whether it is running on **Single** processor machine or **Multi** processor machine.
- Whether it is a **32 bit** machine or **64 bit** machine
- **Read** and **Write** speed of the machine.
- The time it takes to perform **arithmetic operations, logical operation, return value and assignment operations,** etc.
- **Input** data

Calculating Time Complexity

- When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent.

Example, Calculating the Time Complexity required for the following given code

```
int sum(int a, int b) {
return a+b;
}
```

	Time Required	
To **calculate** a+b	1 Unit of time	
For **returning** a+b	1 Unit of time	
Total	**2 Units of time**	

Calculating Time Complexity

- When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent.

Example, Calculating the Time Complexity required for the following given code

```
int sum(int a, int b) {
return a+b;
}
```

	Time Required	*
To **calculate** a+b	1 Unit of time	1 Secs
For **returning** a+b	1 Unit of time	1 Secs
Total	**2 Units of time**	**2 Secs**

*Hypothetical approximation of time

Calculating Time Complexity

- **When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent.**

Example:

```
int sum(int a, int b) {
return a+b; }
```

This Code requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b. That means for all input values, it requires same amount of time i.e. 2 units.

If any program requires fixed amount of time for all input values then its time complexity is said to be **Constant Time Complexity.**

Calculating Time Complexity

Example, Calculate Time complexity for the following given code:

```
int fun(int A[], int n) {
int sum = 0, i;
for(i = 0; i < n; i++)
     sum = sum + A[i];
return sum;
}
```

Calculating Time Complexity

```
int sumOfList(int A[], int n) {
int sum = 0, i;
for(i = 0; i < n; i++)  sum = sum + A[i];
 return sum;  }
```

For the above code, time complexity can be calculated as follows...

	Cost or Number Operations in the Statement		
int sum = 0, i;	1 (sum=0 initializing sum with zero)		
for(i = 0; i < n; i++)			
sum = sum + A[i];			
return sum;			

Calculating Time Complexity

```
int sumOfList(int A[], int n) {
int sum = 0, i;
for(i = 0; i < n; i++)  sum = sum + A[i];
 return sum;  }
```

For the above code, time complexity can be calculated as follows...

	Cost or Number Operations in the Statement		
int sum = 0, i;	1 (initializing zero to sum)		
for(i = 0; i < n; i++)	1 + 1 + 1 (i=0, i<n, i++)		
sum = sum + A[i];			
return sum;			

Calculating Time Complexity

```
int sumOfList(int A[], int n) {
int sum = 0, i;
for(i = 0; i < n; i++)  sum = sum + A[i];
 return sum;  }
```

For the above code, time complexity can be calculated as follows...

	Cost or Number Operations in the Statement		
int sum = 0, i;	1 (initializing zero to sum)		
for(i = 0; i < n; i++)	1+1+1 (i=0, i<n, i++)		
sum = sum + A[i];	1+ 1 (Addition and Assigning result to sum)		
return sum;	1 (returning sum)		

Calculating Time Complexity

```
int sumOfList(int A[], int n) {
int sum = 0, i;
for(i = 0; i < n; i++)  sum = sum + A[i];
 return sum;  }
```

For the above code, time complexity can be calculated as follows...

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**	
int sum = 0, i;	1	**1**	
for(i = 0; i < n; i++)	1+1+1		
sum = sum + A[i];	1+ 1		
return sum;	1		

Calculating Time Complexity

```
int sumOfList(int A[], int n) {
int sum = 0, i;
for(i = 0; i < n; i++)  sum = sum + A[i];
 return sum;  }
```

For the above code, time complexity can be calculated as follows...

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**	
int sum = 0, i;	1	**1**	
for(i = 0; i < n; i++)	1+1+1	**1+(n+1)+n** (i=0 gets executed one time, i<n gets executed (n+1) times, i++ gets executed n times	
sum = sum + A[i];	1+ 1		
return sum;	1		
		Total	

Calculating Time Complexity

```
int sumOfList(int A[], int n) {
int sum = 0, i;
for(i = 0; i < n; i++)  sum = sum + A[i];
 return sum;  }
```

For the above code, time complexity can be calculated as follows...

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**	
int sum = 0, i;	1	**1**	
for(i = 0; i < n; i++)	1+1+1	**1+(n+1)+n**	
sum = sum + A[i];	1+ 1	**n + n**	
return sum;	1	**1**	

Calculating Time Complexity

```
int sumOfList(int A[], int n) {
int sum = 0, i;
for(i = 0; i < n; i++)  sum = sum + A[i];
 return sum;  }
```

For the above code, time complexity can be calculated as follows...

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**	**Total**
int sum = 0, i;	1	**1**	**1**
for(i = 0; i < n; i++)	1+1+1	**1+(n+1)+n**	**2n+2**
sum = sum + A[i];	1+ 1	**n + n**	**2n**
return sum;	1	**1**	**1**
		Running Time T(n)	**4n+4**

Calculating Time Complexity (Contd….

- For the calculation done in previous slide
  Cost is the amount of computer time required for a single operation in each line.
  Repetition is the amount of computer time required by each operation for all its repetitions.
  Total is the amount of computer time required by each operation to execute.

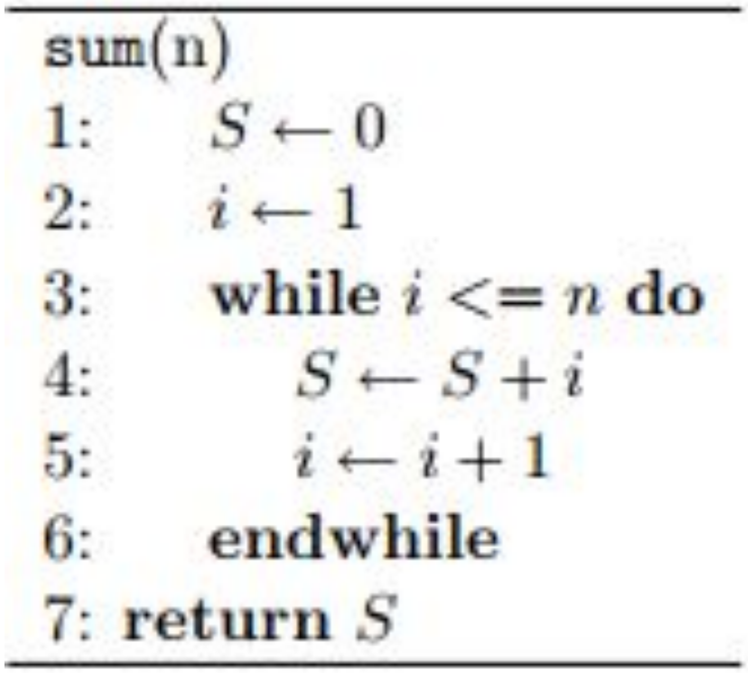
  So above code requires **'4n+4' Units** of computer time to complete the task. Here the exact time is not fixed. And it changes based on the **n** value. If we increase the **n** value then the time required also increases linearly.

  Totally it takes '4n+4' units of time to complete its execution and it is *Linear Time Complexity*.

- If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be **Linear Time Complexity**

Test Your Knowledge

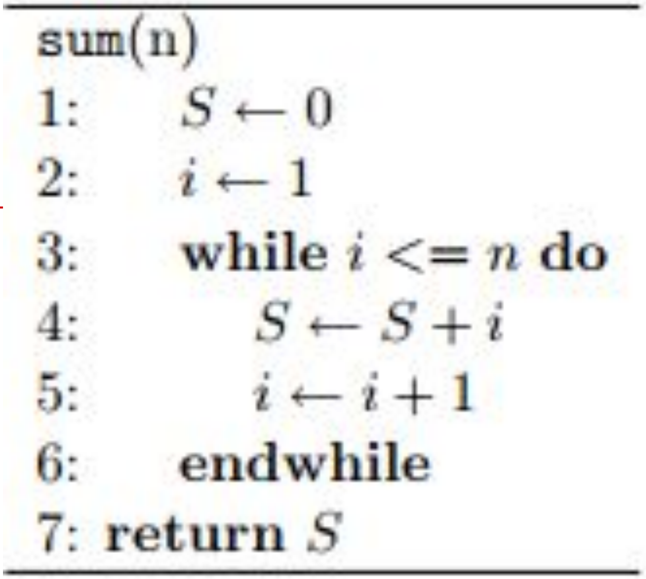
Find Time Complexity of
the given Algorithm



```
sum(n)
1:    $S \leftarrow 0$
2:    $i \leftarrow 1$
3:    while $i <= n$ do
4:        $S \leftarrow S + i$
5:        $i \leftarrow i + 1$
6:    endwhile
7: return $S$
```

Test Your Knowledge

Find Time Complexity of the given Algorithm

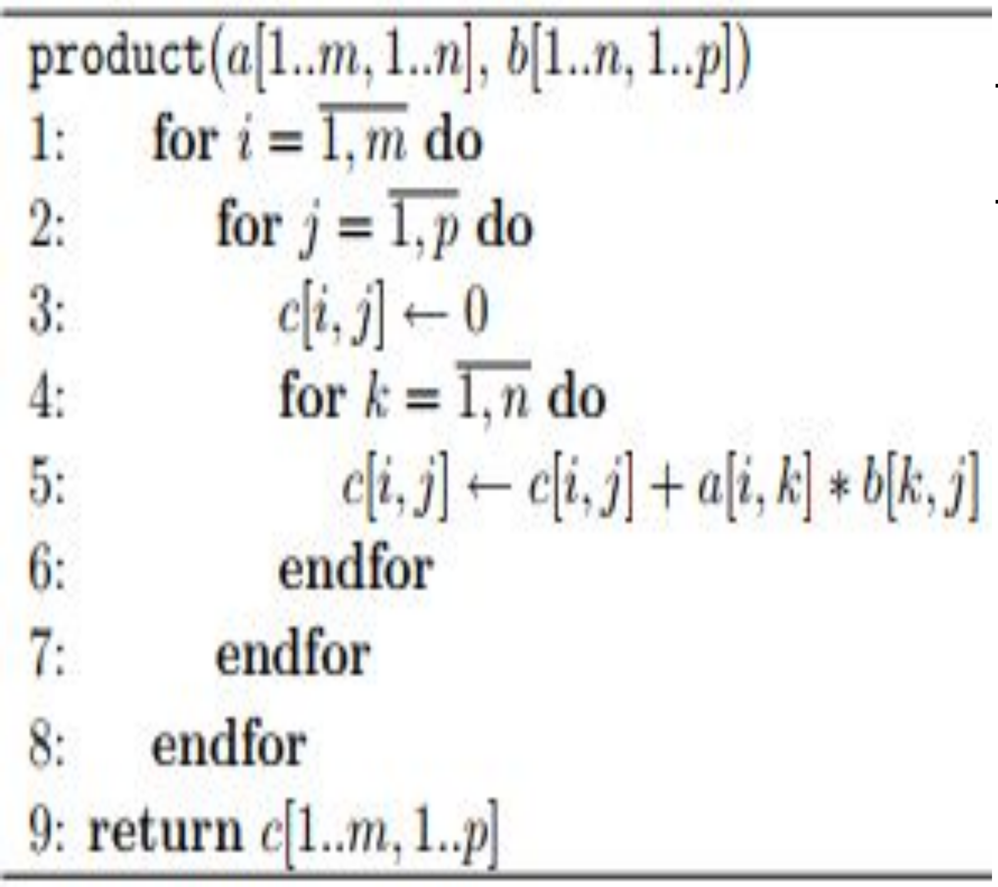


```
sum(n)
1:    $S \leftarrow 0$
2:    $i \leftarrow 1$
3:    while $i <= n$ do
4:        $S \leftarrow S + i$
5:        $i \leftarrow i + 1$
6:    endwhile
7: return $S$
```

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**	**Total**
1	1	**1**	**1**
2	1	**1**	**1**
3	1	**n+1**	**n+1**
4	1+1	**n+n**	**2n**
5	1+1	**n+n**	**2n**
7	1	**1**	**1**
		Running Time T(n)	**5n+4**

Test Your Knowledge

- Find Time Complexity of the following Algorithm

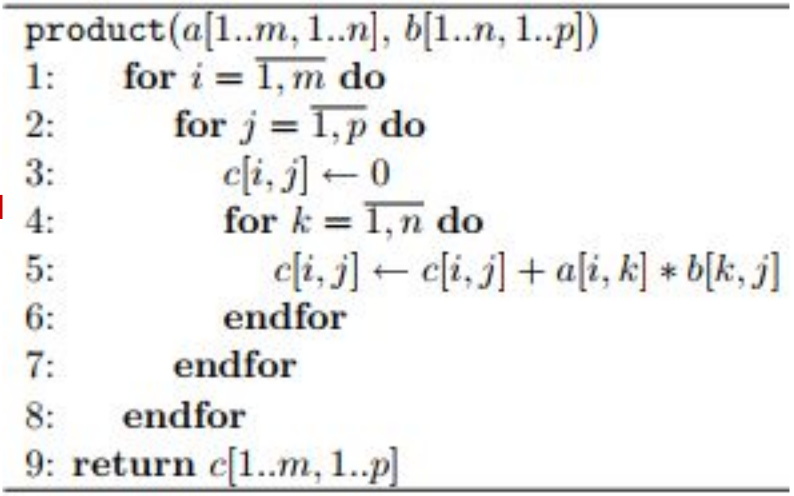


```
product($a[1..m, 1..n]$, $b[1..n, 1..p]$)
    for $i = \overline{1, m}$ do
        for $j = \overline{1, p}$ do
            $c[i, j] \leftarrow 0$
            for $k = \overline{1, n}$ do
                $c[i, j] \leftarrow c[i, j] + a[i, k] * b[k, j]$
            endfor
        endfor
    endfor
return $c[1..m, 1..p]$
```

```
int product(int a[m][n], int b[n][p]){
for(i=1;i<=m;i++){

for(j=1;j<=p;j++){
     c[i][j]=0;

      for(k=1;k<=n;k++){
      c[i][j]=c[i][j]+a[i][k]*b[k][j]

      }

     }

    }
return c
}
```

Test Your Knowledge

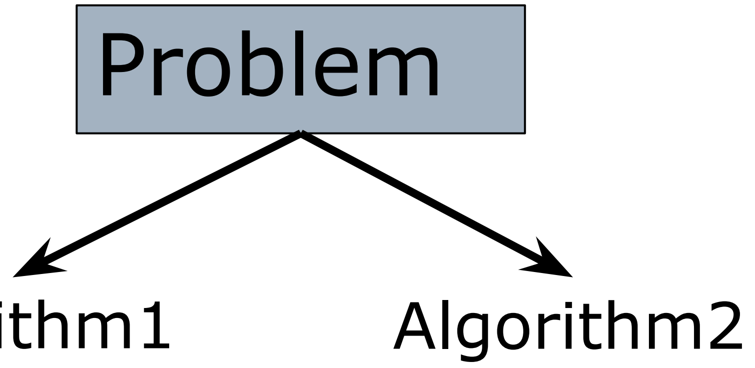
Find Time Complexity for the given Algorithm



```
product($a[1..m, 1..n]$, $b[1..n, 1..p]$)
1:    for $i = \overline{1, m}$ do
2:        for $j = \overline{1, p}$ do
3:            $c[i, j] \leftarrow 0$
4:            for $k = \overline{1, n}$ do
5:                $c[i, j] \leftarrow c[i, j] + a[i, k] * b[k, j]$
6:            endfor
7:        endfor
8:    endfor
9: return $c[1..m, 1..p]$
```

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**	**Total**	
1	1+1+1	**1+(m+1)+m**	**2m+2**	**2m+2**
2	1+1+1	**(1+(p+1)+p)m**	**(2p+2)m**	**2pm+2m**
3	1	**(p)m**	**pm**	**pm**
4	1+1+1	**((1+(n+1)+n)p)m**	**((2n+2)p) m**	**2npm+ 2pm**
5	1+1+1	**((n+n+n)p)m**	**((3n)p)m**	**3npm**
9	1	**1**	**1**	**1**
			Running Time T(n)	**5npm+ 5pm+4m +3**

Given two algorithms for a task, how do we find out which one is better?



One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.
1) It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
2) It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

Question

What is the meaning of the notation **T(n)** w.r.t analysis of algorithms ?

Answer

- $T_A(\mathbf{n})$ = Maximum time taken (or Number of Machine operations needed) by the algorithm A to solve **input of size n**.
- Input size refers to number of values in the data set. Example: Say ten lakh Aadhar card numbers has to be sorted then **input size n** refers to 10,00,000
- $T_A(\mathbf{n})$ is the measure of **Goodness** of Algorithm A

T(n)

Expression we get for T(n) may not be of great consequence for real Computers /Computations because it various from one machine architecture to another machine architecture

Example: Program 1

```
#include <stdio.h>
main(){
int n,temp;

scanf("%d",&n)

temp=10*30;
}
```

Example: Program 1

```
#include <stdio.h>
main(){
int n,temp;

scanf("%d",&n)

temp=10*30;
}
```

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**	**Total**
temp=10*30;	1+1	1+1	**2**
		Running Time T(n)	**2**

Example: Program 1

```
#include <stdio.h>
main(){
int n,temp;

scanf("%d",&n)

temp=10*30;
}
```

Running Time T(n)=2

Constant Time

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**	**Total**
temp=10*30;	1+1	**1+1**	**2**
		Running Time T(n)	**2**

Example: Program 2

```
#include <stdio.h>
main(){
int i, n,temp;
scanf("%d",&n)
for(i=0; i < n; i++) {
        temp=10*30;
             }
}
```

Example: Program 2

```
#include <stdio.h>
main(){
int i, n,temp;
scanf("%d",&n)
for(i=0; i < n; i++) {
        temp=10*30;
              }
}
```

	Cost or Number of Operations in the Statement	**Repetitions or No. of Times of Execution**	**Total**
for(i=0; i < n; i++)	1+1+1	**1+(n+1)+n**	**2n+2**
temp=10*30;	1+1	**n+n**	**2n**
		Running Time T(n)	**4n+2**

T(n) for different values of n

```
#include <stdio.h>
main(){
int i, n,temp;
scanf("%d",&n)
for(i=0; i < n; i++) {
          temp=10*30;
                }  }
```

Running Time T(n)=4n+2

n	T(n)=4n+2	On Computer 1 Time taken(in Secs)	On Computer 2 Time taken (in Secs)
10	42	??	??
20	82	??	??
1000	4002	??	??
50000	20002	??	??

T(n) for different values of n

Running Time T(n)=4n+2

```
#include <stdio.h>
#include <time.h>
main(){
long int n,i; int temp;
clock_t start, end;

scanf("%ld",&n)
start=clock(); //clock() give the current time of the system in clock ticks
for(i=0; i < n; i++) {
          temp=10*30;
                }
end=clock();
printf("Time take %f in Secs",(((double)(end-start))/CLOCKS_PER_SEC));
}
```

Delay

- for (c = 1; c <= 5000; c++) for (d = 1; d <= 5000; d++) { }
  - void delay(int number_of_seconds) {
  - // Converting time into milli_seconds
  - int milli_seconds = 1000 * number_of_seconds; // Storing start time clock_t start_time = clock(); // looping till required time is not achieved while (clock() < start_time + milli_seconds) ; }
- #include<stdio.h> delay(3);

T(n) for different values of n

```
#include <stdio.h>
main(){
int i, n,temp;
scanf("%d",&n)
for(i=0; i < n; i++) {
          temp=10*30;
                }  }
```

n	T(n)=4n+2	On Computer 1 Time taken(in Secs)	On Computer 2 Time taken (in Secs)
10	42	0.000002	
20	82	0.000003	
1000	4002	0.000010	
50000	200002	0.000395	

T(n) for different values of n

```
#include <stdio.h>
main(){
int i, n,temp;
scanf("%d",&n)
for(i=0; i < n; i++) {
          temp=10*30;
                }  }
```

n	T(n)=4n+2	On Computer 1 Time taken(in Secs)	On Computer 2 Time taken (in Secs)
10	42	0.000002	0.000002
20	82	0.000003	0.000002
1000	4002	0.000010	0.000006
50000	200002	0.000395	0.000252

T(n) for different values of n

```
#include <stdio.h>
main(){
int i, n,temp;
scanf("%d",&n)
for(i=0; i < n; i++) {
          temp=10*30;
                } }
```

Running Time T(n)=4n+2

Linear Time

n	T(n)=4n+2	On Computer 1 Time taken(in Secs)	On Computer 2 Time taken (in Secs)
10	42	0.000002	0.000002
20	82	0.000003	0.000002
1000	4002	0.000010	0.000006
50000	200002	0.000395	0.000252

Example: Program 3

```
#include <stdio.h>
main(){  int i,j,n,temp;
scanf("%d", &n)
for(i=0; i < n; i++) {
      for(j=0; j < n;j++) {
            temp=10*30;
             }  } }
```

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**	**Total**
for(i=0; i < n; i++)	1+1+1	1+(n+1)+n	$2n+2$
for(j=0; j < n; j++)	1+1+1	(1+(n+1)+n)n	$2n^2+2n$
temp=10*30;	1+1	(n+n)n	$2n^2$
		Running Time T(n)	$4n^{2+}4n+2$

T(n) for different values of n

```
#include <stdio.h>
main(){  int i,j,n,temp;
scanf("%d", &n)
for(i=0; i < n; i++) {
     for(j=0; j < n;j++) {
          temp=10*30;
          }  } }
```

Running Time T(n)=$4n^{2+}4n+2$

Quadratic Time

n	T(n)=$4n^2+4n+2$	On Computer 1 Time taken(in Secs)	On Computer 2 Time taken (in Secs)
10	442	0.000003	0.000002
20	1682	0.000010	0.000007
1000	4004002	0.008255	0.005062
50000	10000200002	5.766243	4.765878

Rate of Growth or Order of Growth

Order of growth in algorithm means how the time for computation increases when you increase the input size. It really matters when your input size is very large.

Order of growth provide only a crude description of the behavior of a process.

Algorithms analysis is all about understanding growth rates. That is as the amount of data gets bigger, how much more resource will my algorithm require? Typically, we describe the resource growth rate of a piece of code in terms of a function.

Order of Growth: Linear vs Quadratic

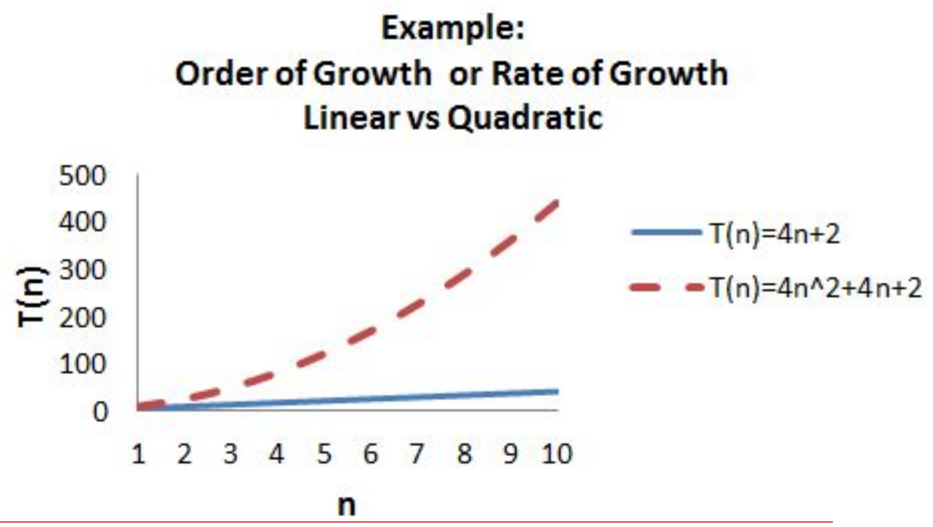
n	T(n)=4n+2	T(n)=4n^2+4n+2
1	6	10
2	10	26
3	14	50
4	18	82
5	22	122
6	26	170
7	30	226
8	34	290
9	38	362
10	42	442

Example

Order of Growth: Linear vs Quadratic

n	T(n)=4n+2	T(n)=4n^2+4n+2
1	6	10
2	10	26
3	14	50
4	18	82
5	22	122
6	26	170
7	30	226
8	34	290
9	38	362
10	42	442

Example



Example: Program 4

```
#include <stdio.h>
main(){  int i,j,k, n,temp;
scanf("%d", &n)
for(i=0; i < n; i++) {
      for(j=0; j < n;j++) {
             for(k=0; k < n;k++) {
                  temp=10*30;
             }  } } }
```

Example: Program 4

```
#include <stdio.h>
main(){  int i,j,k, n,temp;
scanf("%d", &n)
for(i=0; i < n; i++) {
      for(j=0; j < n;j++) {
             for(k=0; k < n;k++) {
                  temp=10*30;
             }  } } }
```

Running Time T(n)=$4n^3+4n^2+4n+2$

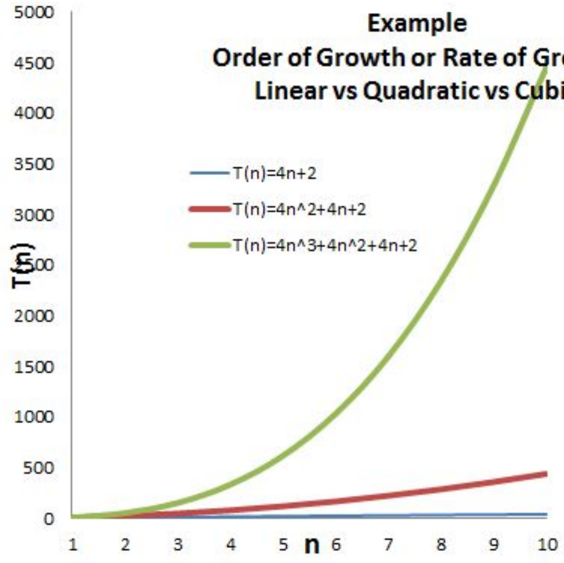
Cubic Time

	Cost or Number Operations in the Statement	**Repetitions or No. of Times of Execution**	**Total**
for(i=0; i < n; i++)	1+1+1	$1+(n+1)+n$	$2n+2$
for(j=0; j < n; j++)	1+1+1	$(1+(n+1)+n)n$	$2n^2+2n$
for(k=0; k < n;k++)	1+1+1	$((1+(n+1)+n)n)n$	$2n^3+2n^2$
temp=10*30;	1+1	$((n+n)n)n$	$2n^3$
		Running Time T(n)	$4n^3+4n^2+4n+2$

Order of Growth (or Rate of Growth): Linear vs Quadratic vs Cubic

Example

n	T(n)=4n+2	T(n)=4n^2+4n+2	T(n)=4n^3+4n^2+4n+2
1	6	10	14
2	10	26	58
3	14	50	158
4	18	82	338
5	22	122	622
6	26	170	1034
7	30	226	1598
8	34	290	2338
9	38	362	3278
10	42	442	4442



Interpretation of T(n)

- What is important is "**form (or shape)** of T(n)" i.e.., whether T(n) is Linear, Quadratic, Cubic..etc.
- Using the expression of T(n) we may not be able to give exact estimate but we can interpret the **behavior of the algorithm** when implemented on any computer.
- Analyzing the behavior of the algorithm for **LARGE n** is important.(i.e., as n tends to infinity n -> ∞)

Question

Consider, you are given with 10 Aadhaar card numbers and you are asked to sort this numbers in Ascending order. Assume Aadhaar card numbers are available in an Notepad file stored on computer memory. Which of the following strategy you will use:

a. Sort by hand (or mentally) and update the file
b. Sort by writing a program

Question

Consider, you are given with 1000 Aadhaar card numbers and you are asked to sort this numbers in Ascending order. Assume Aadhaar card numbers are available in an database file stored on computer memory. Which of the following strategy you will use:

a. Sort by hand (or mentally) and update the file
b. Sort by writing a program

Question

Consider, you are given with 10,000 Aadhaar card numbers and you are asked to sort this numbers in Ascending order. Assume, Aadhaar card numbers are available in an database file stored on computer memory.

You are given with two sorting algorithms, say the efficiency of Algorithm1 is $T_{A1}(n)=4n+2$ and efficiency of Algorithm2 is $T_{A2}(n)=4n^2+2$

Which of the following strategy you will use:

a. Write a program to sort by implementing Algorithm1
b. Write a program to sort by implementing Algorithm2

Answer

Consider, you are given with 10,000 Aadhaar card numbers and you are asked to this numbers in Ascending order. Assume, Aadhaar card numbers are available in an database file stored on computer memory.

You are given with two sorting algorithms, say the efficiency of Algorithm1 is $T_{A1}(n)=4n+2$ and efficiency of Algorithm2 is $T_{A2}(n)=4n^2+2$

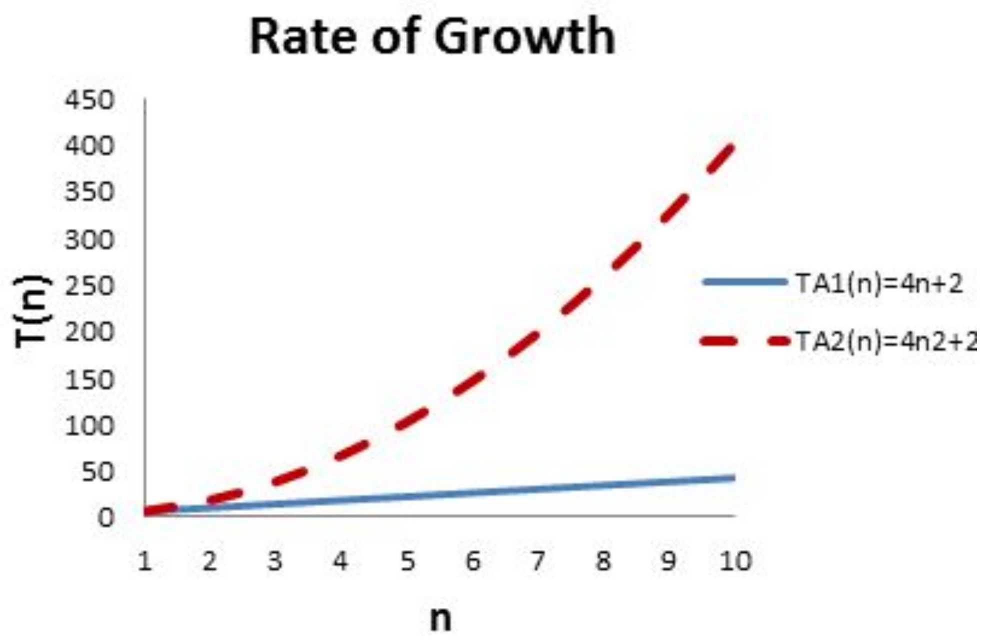
Which of the following strategy you will use:

a. **Write a program to sort by implementing Algorithm1**

b. Write a program to sort by implementing Algorithm2

We can choose eitherAlgo1 or Algo2 if it is one time sorting. And also on modern computer Sorting ten thousand numbers Will not take much time because it will be done in fraction of seconds

n	$T_{A1}(n)=4n+2$	$T_{A2}(n)=4n^2+2$
1	6	6
2	10	18
3	14	38
4	18	66
5	22	102
6	26	146
7	30	198
8	34	258
9	38	326
10	42	402



Question

Consider, you are given with 10,00,000 Aadhaar card numbers and you are asked sort to this numbers in Ascending order. Assume, Aadhaar card numbers are available in an database file stored on computer memory.

You are given with two sorting algorithms, say the efficiency of Algorithm1 is $T_{A1}(n)=10^4n$ and efficiency of Algorithm2 is $T_{A2}(n)=n^2$

Which of the following strategy you will use:

a. Write a program to sort by implementing Algorithm1
b. Write a program to sort by implementing Algorithm2

Answer

Consider, you are given with 10,00,000 Aadhaar card numbers and you are asked sort to this numbers in Ascending order. Assume, Aadhaar card numbers are available in an database file stored on computer memory.

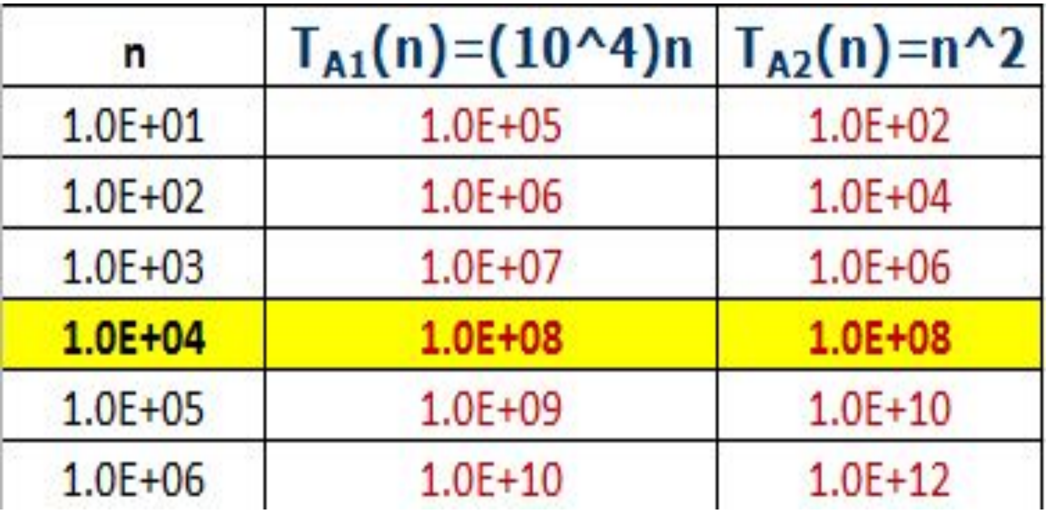
You are given with two sorting algorithms, say the

efficiency of Algorithm1 is $T_{A1}(n)=10^4n$ and

efficiency of Algorithm2 is $T_{A2}(n)=n^2$

Which of the following strategy you will use:

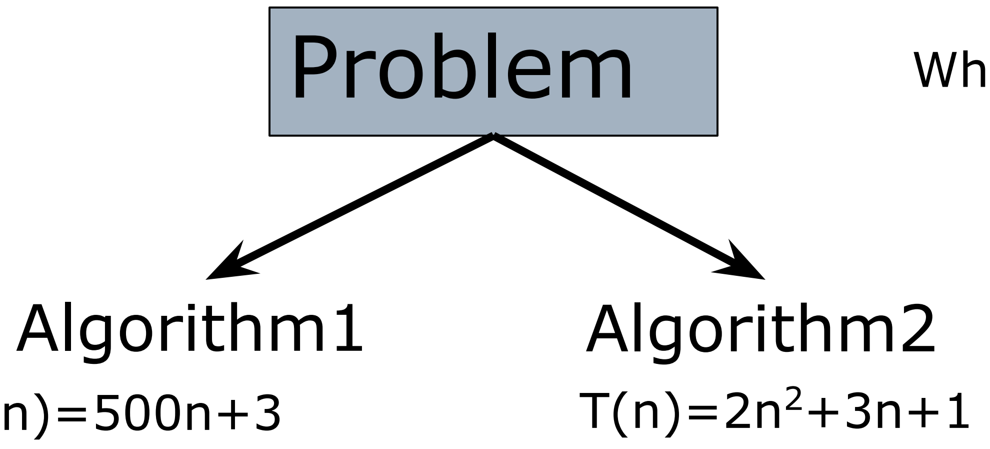
a. Write a program to sort by implementing Algorithm1
b. Write a program to sort by implementing Algorithm2



n	$T_{A1}(n)=(10\text{^}4)n$	$T_{A2}(n)=n\text{^}2$
1.0E+01	1.0E+05	1.0E+02
1.0E+02	1.0E+06	1.0E+04
1.0E+03	1.0E+07	1.0E+06
1.0E+04	**1.0E+08**	**1.0E+08**
1.0E+05	1.0E+09	1.0E+10
1.0E+06	1.0E+10	1.0E+12

For $n<=10^4$,$10^4n > n^2$ *Algo2* is better
But for $n> 10^4$ *Algo1* is better

Question



Which algorithm to use ?

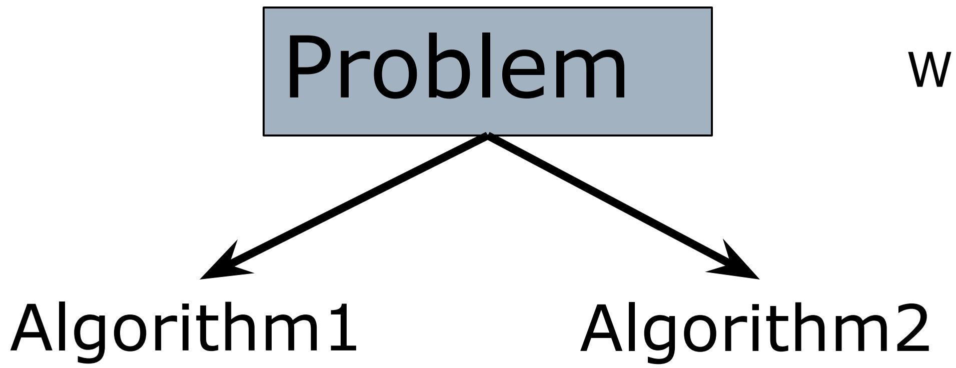
Question

$$\begin{tikzcd} & \boxed{\text{Problem}} \arrow[dl] \arrow[dr] & \\ \text{Algorithm1} & & \text{Algorithm2} \end{tikzcd}$$

$T(n)=10n^3+5n^2+17$ $\qquad$ $T(n)=2n^3+3n+79$

Which algorithm to use ?

Question



Which algorithm to use ?

$T(n)=10n^3+5n^2+17$ $T(n)=2n^3+3n+79$

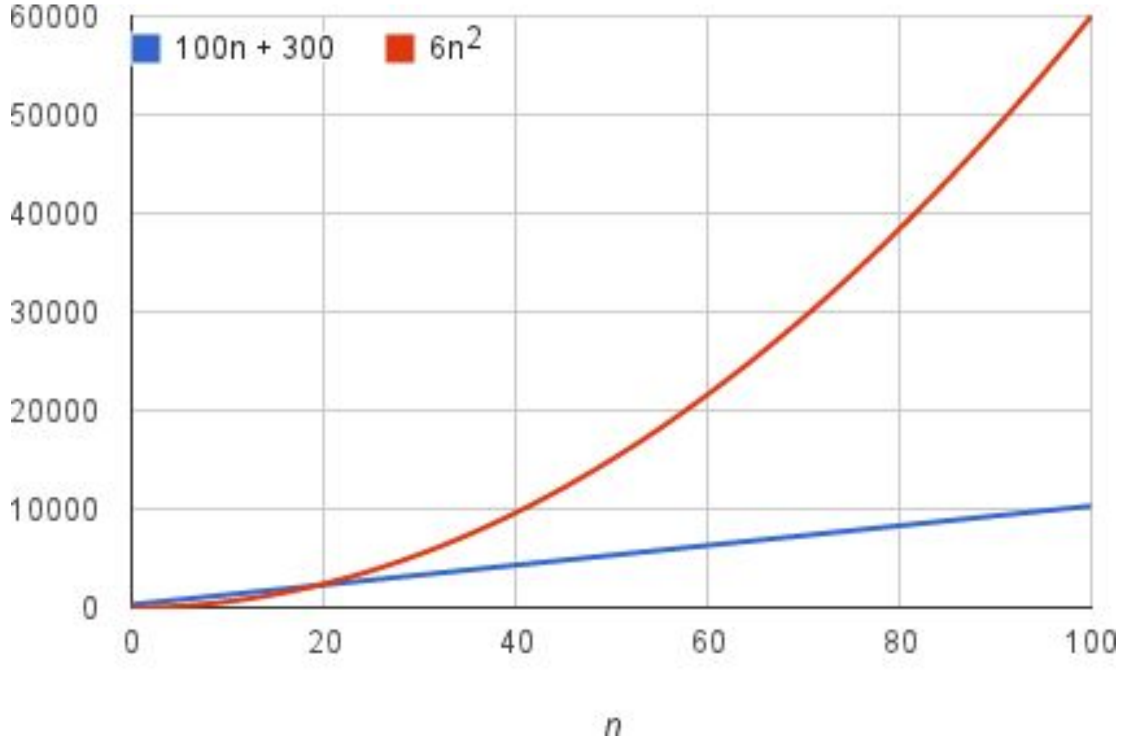
Answer:
The above two time complexities are tedious to be judged.
Hence we will go with **approximating the time complexities** i.e., finding Out the class to which the algorithm belongs because as n tends to infinity ($n \rightarrow \infty$) i.e., when n takes large values the value of $(5n^2+17)$
and the value of $(3n+79)$ will go out. Therefore we will be worrying about $10n^3$ and $2n^3$

Example

Consider T(n)= $\mathbf{6n^2 + 100n + 300}$

Example

Consider T(n)= **$6n^2 + 100n + 300$**



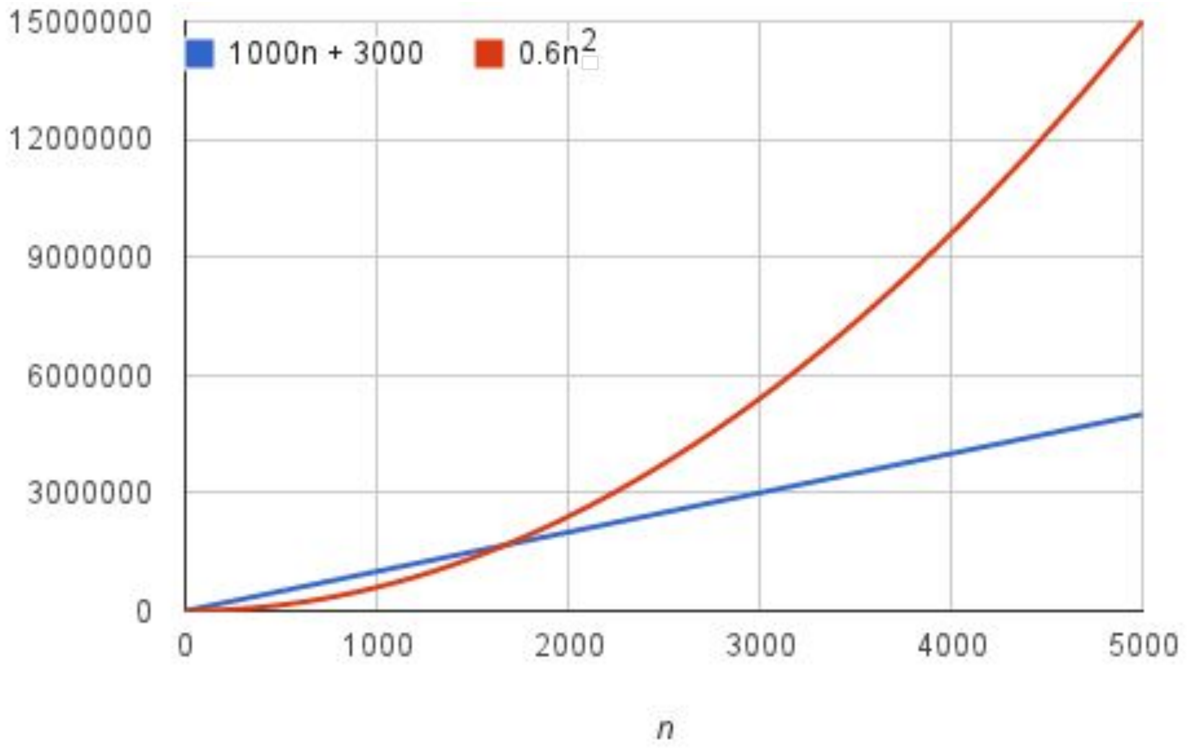
The $6n^2$ term becomes larger than the remaining terms, 100n + 300, once n becomes large enough, 20 in this case.

Example

Consider $T(n) = \mathbf{0.6n^2 + 1000n + 3000}$

Example

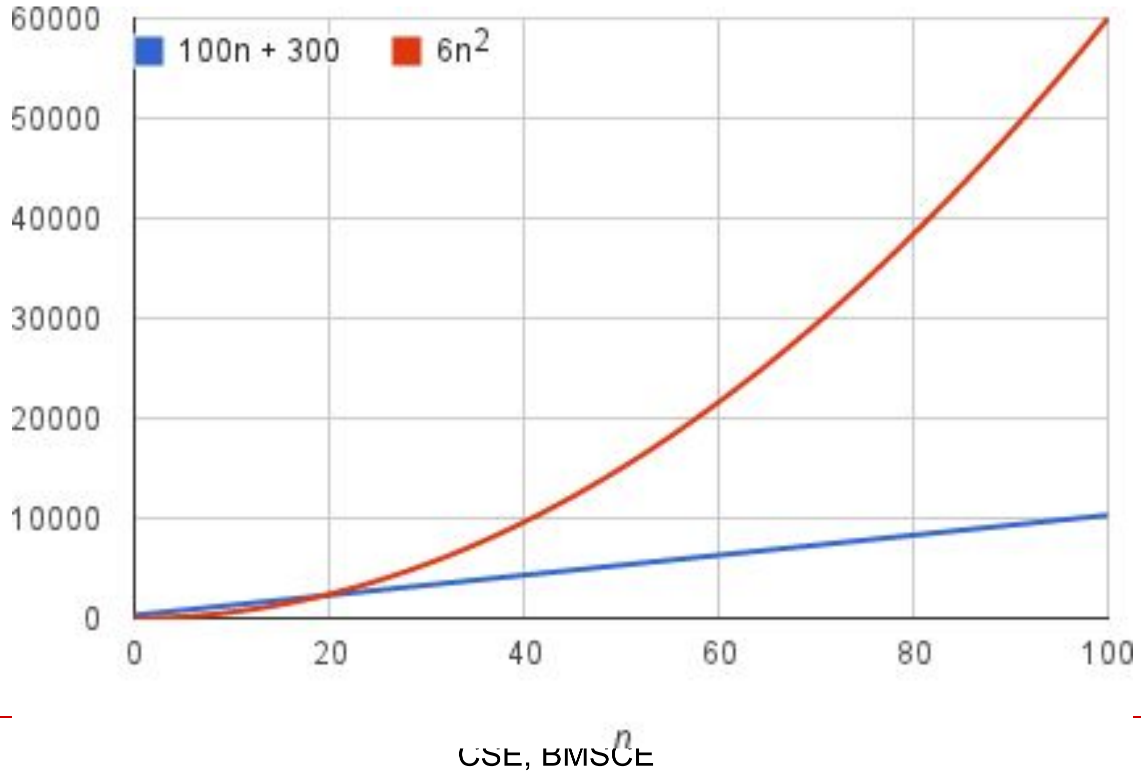
Consider T(n)= $\mathbf{0.6n^2 + 1000n + 3000}$



The $0.6n^2$ term becomes larger than the remaining terms, 1000n + 3000, once n becomes large enough, 1700 in this case.

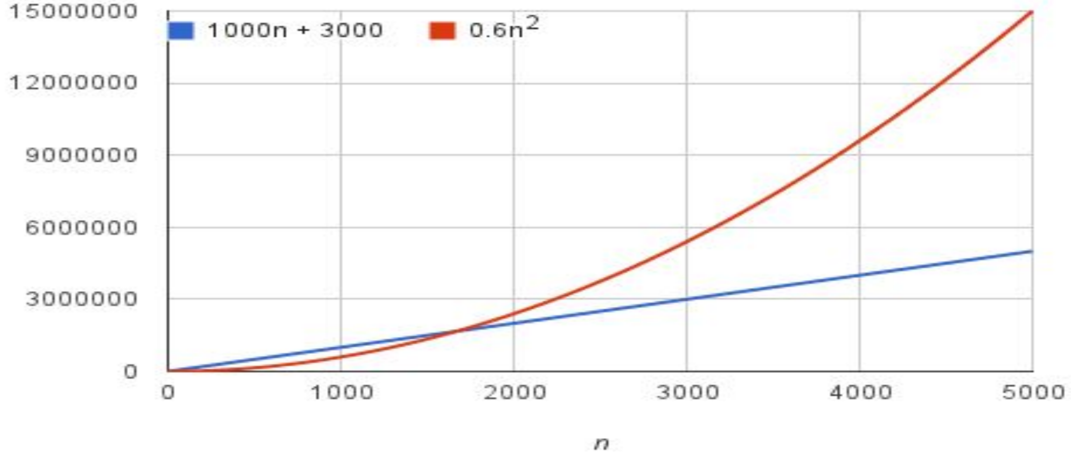
Explanation

For example, suppose that an algorithm, running on an input of size n, takes **$6n^2$ + 100n + 300** machine instructions. The $6n^2$ term becomes larger than the remaining terms, 100n + 300, once n becomes large enough, 20 in this case. Here's a chart showing values of **$6n^2$** and **100n + 300** for values of *n* from 0 to 100:



Explanation

We would say that the running time of this algorithm grows as n^2, dropping the coefficient 6 and the remaining terms $100n + 300$. It doesn't really matter what coefficients we use; as long as the running time is $an^2 + bn + c$, for some numbers $a > 0$, b, and c, there will always be a value of n for which an^2 is greater than $bn + c$, and this difference increases as n increases. For example, here's a chart showing values of $0.6n^2$ and $1000n + 3000$ so that we've reduced the coefficient of n^2 by a factor of 10 and increased the other two constants by a factor of 10:



The value of n at which $0.6n^2$ becomes greater than $1000n + 3000$ has increased, but there will always be such a crossover point, no matter what the constants.

What is a Time Complexity/Order of Growth?

- Time Complexity/Order of Growth defines the amount of time taken by any program with respect to the size of the input.
- Time Complexity specifies how the program would behave as the order of size of input is increased. So, Time Complexity is just a function of size of its input.

Some of basic and most common time complexities such as:

- Constant Time Complexity: Constant running time
- Linear Time Complexity *(n)* : Linear running time
- Logarithmic Time Complexity *(log n)* : Logarithmic running time
- Log-Linear Time Complexity *(n log n)* : Log-linear running time
- Polynomial Time Complexity *(n^c)* : Polynomial running time (c is a constant)
- Exponential Time Complexity *(c^n)* : Exponential running time (c is a constant being raised to a power based on size of input)

What is Constant Time Complexity?

- The code that runs in fixed amount of time or has fixed number of steps of execution no matter what is the size of input has constant time complexity. For instance, let’s try and derive a Time Complexity for following code:

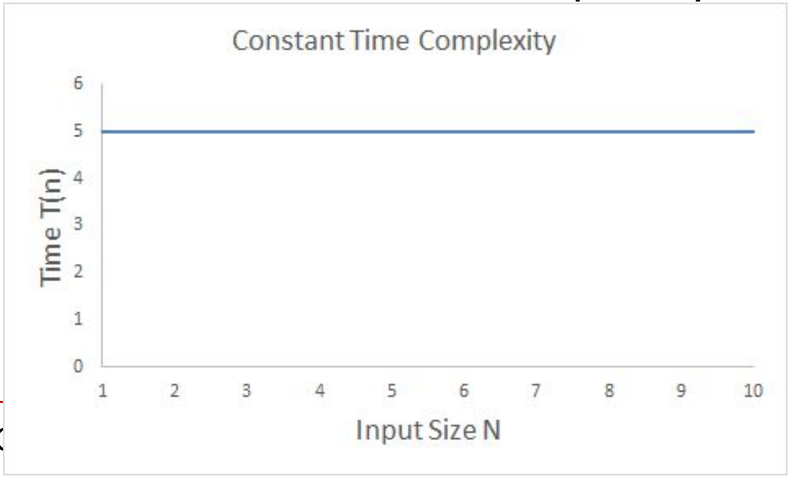
What is Constant Time Complexity?

- The code that runs in fixed amount of time or has fixed number of steps of execution no matter what is the size of input has constant time complexity. For instance, let’s try and derive a Time Complexity for following code:

```
def my_sum(a, b):
        return a+b
```

If we call this function by my_sum(2, 5) it will return 7 in 1 step. That single step of computation is summing a and b. No matter how large is the size of input i.e. a and b is, it will always return the sum in 1 step.

So, the Time Complexity of the above code is a Constant Time Complexity.



What is Linear Time Complexity?

The code whose Time Complexity or Order of Growth increases linearly as the size of the input is increased has Linear Time Complexity.

For instance, let’s see this code which returns the sum of a list.

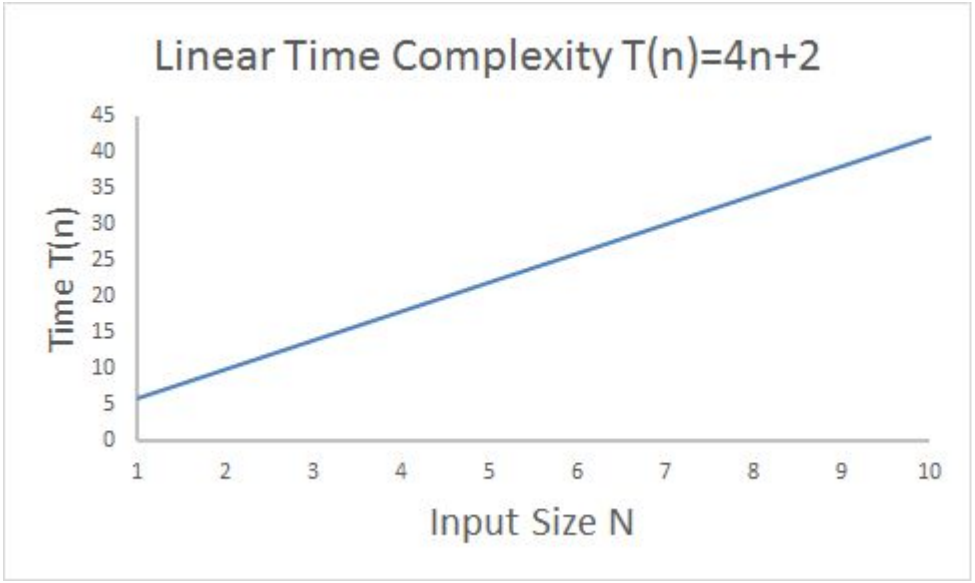
What is Linear Time Complexity?

The code whose Time Complexity or Order of Growth increases linearly as the size of the input is increased has Linear Time Complexity.

For instance, let's see this code which returns the sum of a list.

```
for(i=0; i < n; i++)
    temp=10*30;
```

$T(n) = 4n+2$



What is Logarithmic Time Complexity?

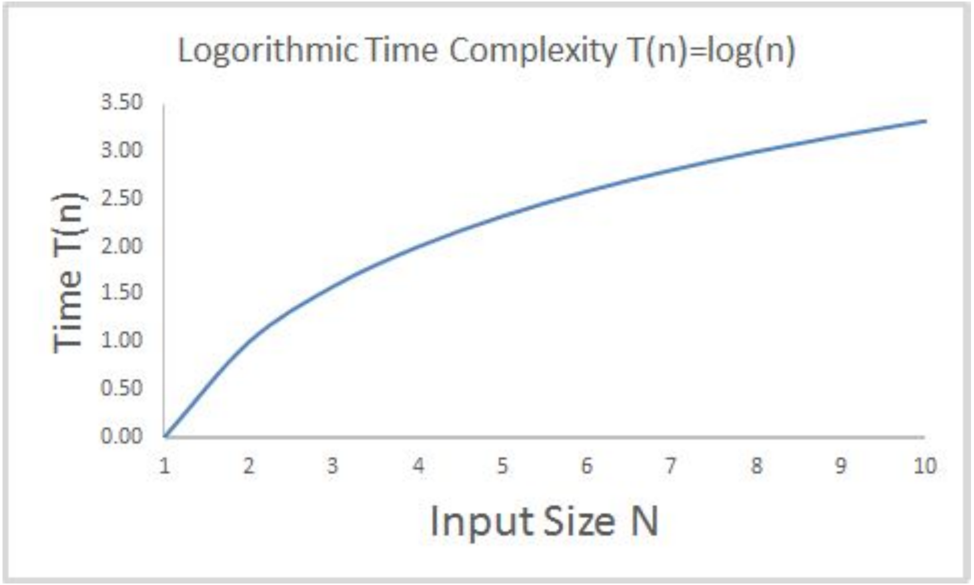
- When the size of input is N but the number of steps to execute the code is log(N), such a code is said to be executing in Logarithmic Time. This definition is quite vague but if we take an example, it will be quite clear.

What is Logarithmic Time Complexity?

- When the size of input is N but the number of steps to execute the code is log(N), such a code is said to be executing in Logarithmic Time. This definition is quite vague but if we take an example, it will be quite clear.
- Let's say we have a very large number which is a power of 2 i.e. we have 2^x. We want to find x. For eg: 64 = 2^6. So x is 6.

```
pow(n){
   x = 0
   while (n > 1){
      n = n/2
      x = x+1}
   return x }
```

T(n) = log(n)



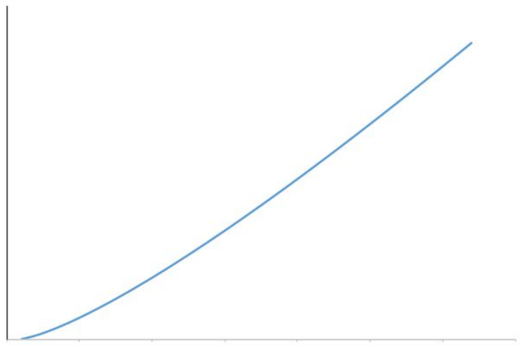
What is Log-Linear Time Complexity?

- When we call a Logarithmic Time Algorithm inside a loop, it would result into a Log-Linear Time Complexity program.
- For example: Let's say we have long sorted list of size N. And we have Q numbers, for each of those Q numbers we have to find the index of it in the given list.

```
for i in Qlist:
    print binary_search(x, search_list)  #This statement is
                                          #executed Q times
```

Analyzing above code, we know that the call to Binary Search function takes (log N) times. We are calling it Q times. Hence the overall time complexity is Q(log N).

Log Linear Time Complexity



What is Polynomial Time Complexity?

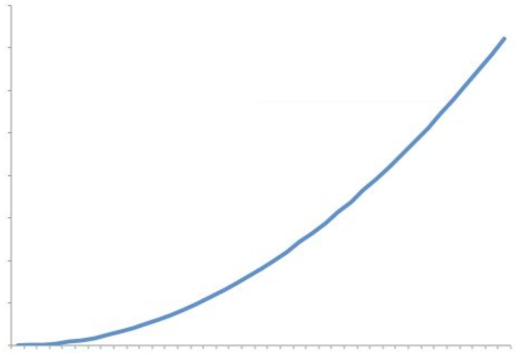
When the computation time increases as function of N raised to some power, N being the size of input. Such a code has Polynomial Time Complexity.

For example, let’s say we have a list of size N and we have nested loops on that list.

```
for i in N:
    for j in N:
        # Some processing
```

In the above code, the processing part is executed N*N times i.e. N^2 times. Such a code has *(N^2)* time complexity.

Quadratic Time Complexity

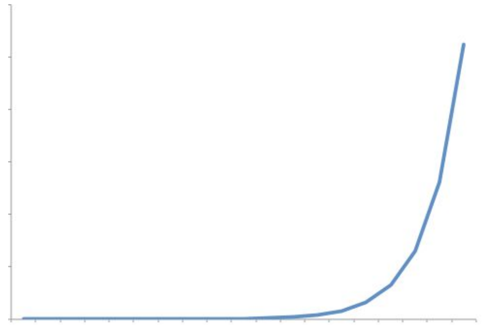


What is Exponential Time Complexity?

- When the computation time of a code increases as function of X^N, N being the size of input. Such a code has Polynomial Time Complexity.
- For example, following recursive code to find Nth fibonacci number has Time Complexity as (**2^N**)

```
def F(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return F(n-1) + F(n-2) # For every call to F, we make 2 more calls to
F itself
```

Exponential Time Complexity



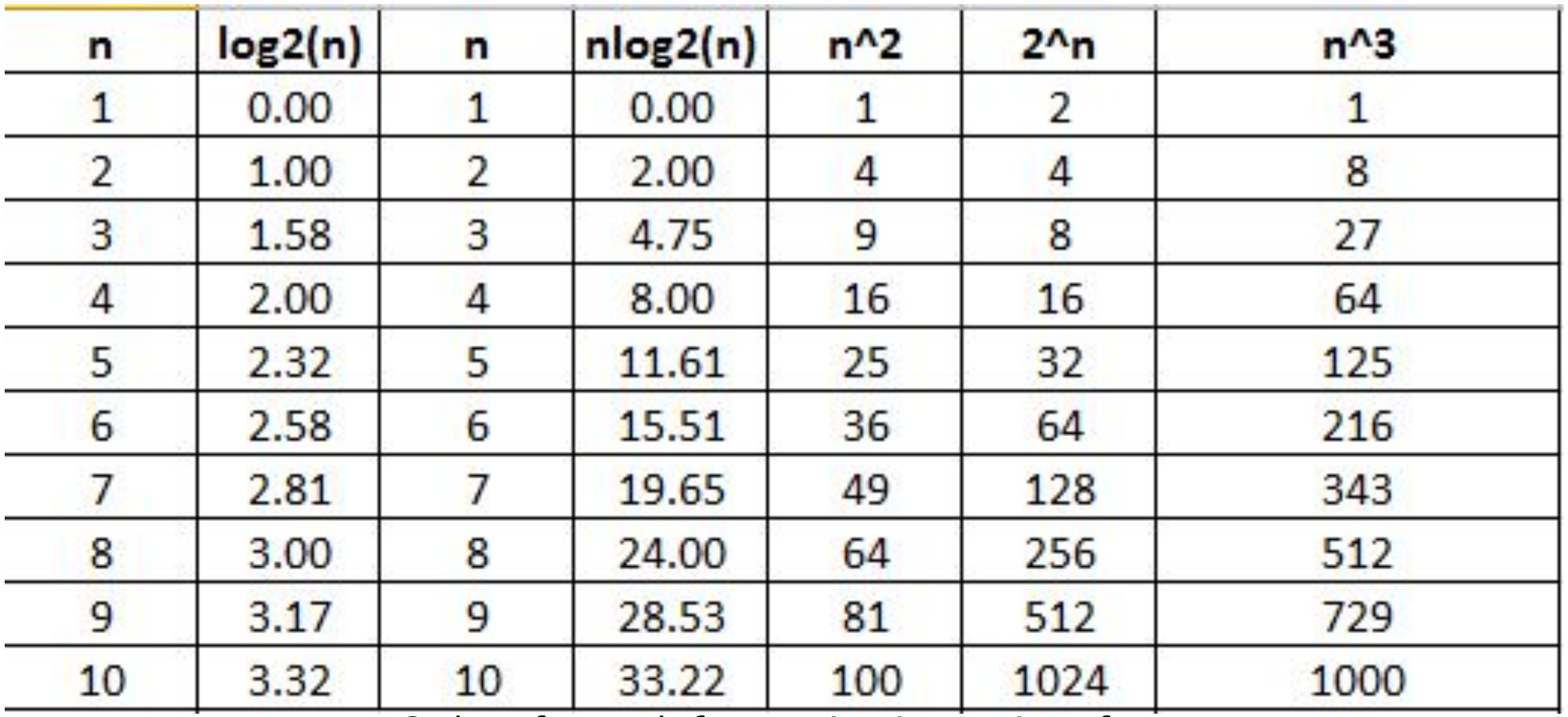
-We consider only leading term in the expression T(n), since lower-order terms are relatively insignificant for large n.

-We are moving from **Actual cost** to **Growth of Cost (or Rate of Growth).**

-We are interested to know what is the term that dominates so that if we arbitrarily keep on increasing *n* that is the term which primarily decide how the computing time will grow.

Order of Growth

Measuring the performance of an algorithm in relation with the input size n is called Order of growth or Rate of Growth



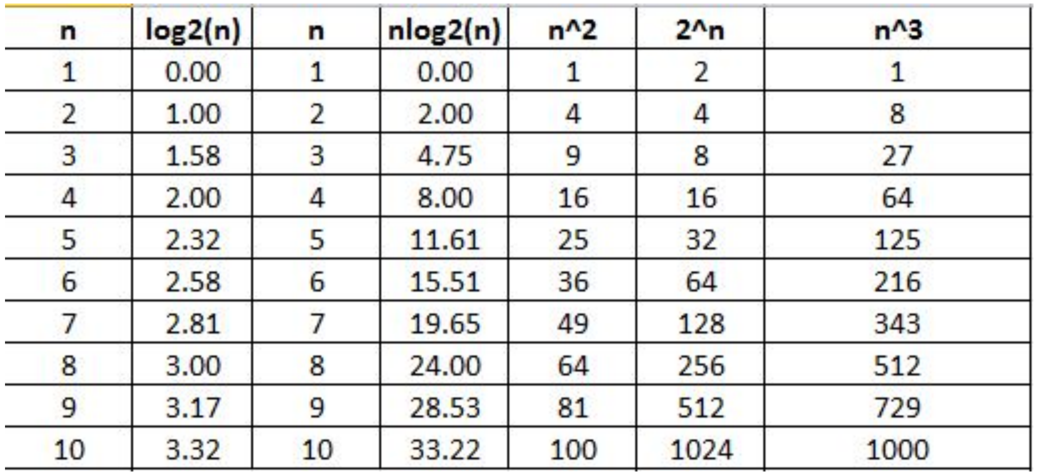
n	log2(n)	n	nlog2(n)	n^2	2^n	n^3
1	0.00	1	0.00	1	2	1
2	1.00	2	2.00	4	4	8
3	1.58	3	4.75	9	8	27
4	2.00	4	8.00	16	16	64
5	2.32	5	11.61	25	32	125
6	2.58	6	15.51	36	64	216
7	2.81	7	19.65	49	128	343
8	3.00	8	24.00	64	256	512
9	3.17	9	28.53	81	512	729
10	3.32	10	33.22	100	1024	1000

Order of growth for varying input size of n

Order of Growth

Measuring the performance of an algorithm in relation with the input size n is called order of growth . Some of the popular order which we will see is:-

- Order 1 : Constant.
- Order log(n) : Logarithmic
- Order (n) : linear
- Order nlog(n): log linear, occurs very often
- order (n ^ C) : polynomial
- order (C ^ n) : exponential



n	log2(n)	n	nlog2(n)	n^2	2^n	n^3
1	0.00	1	0.00	1	2	1
2	1.00	2	2.00	4	4	8
3	1.58	3	4.75	9	8	27
4	2.00	4	8.00	16	16	64
5	2.32	5	11.61	25	32	125
6	2.58	6	15.51	36	64	216
7	2.81	7	19.65	49	128	343
8	3.00	8	24.00	64	256	512
9	3.17	9	28.53	81	512	729
10	3.32	10	33.22	100	1024	1000

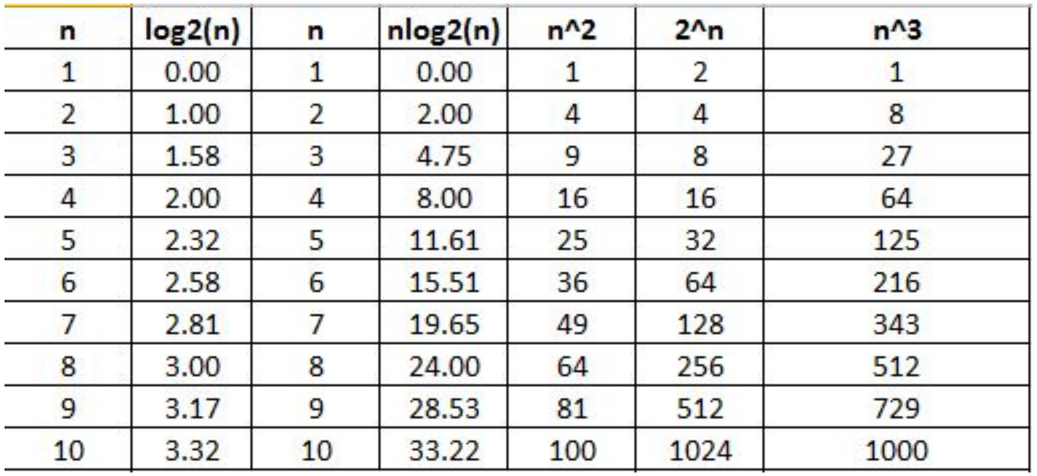
Order of growth for varying input size of n

Order of Growth

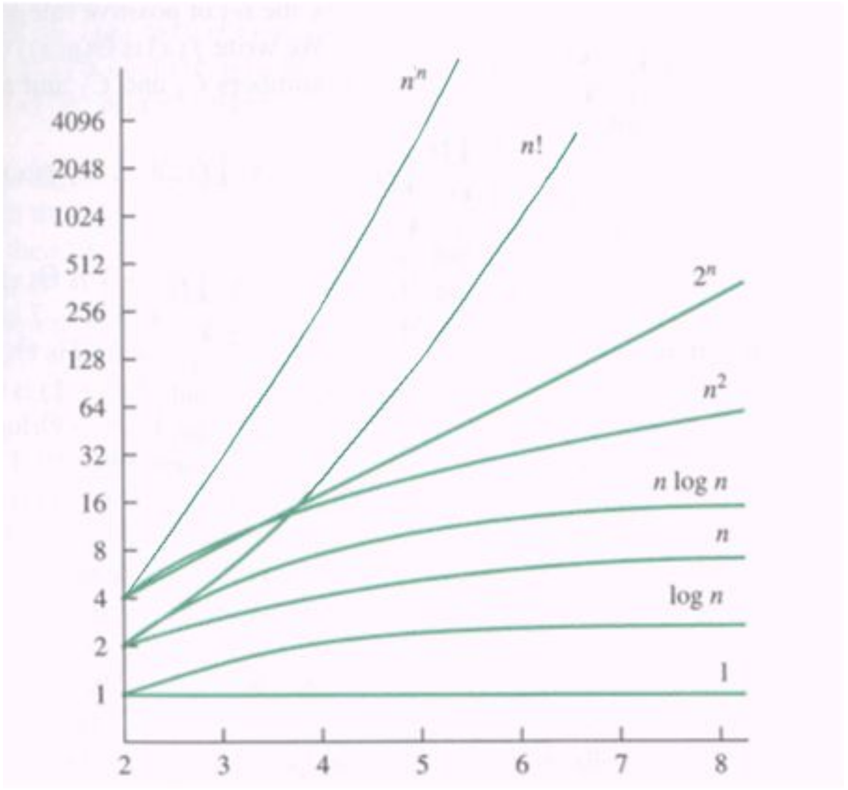
Measuring the performance of an algorithm in relation with the input size n is called order of growth

Some of the popular order which we will see is:-

- Order 1 : Constant.
- Order log(n) : Lograthimic
- Order (n) : liner
- Order nlog(n): log liner, occurs very often
- order (n ^ C) : polynomial
- order (C ^ n) : exponential



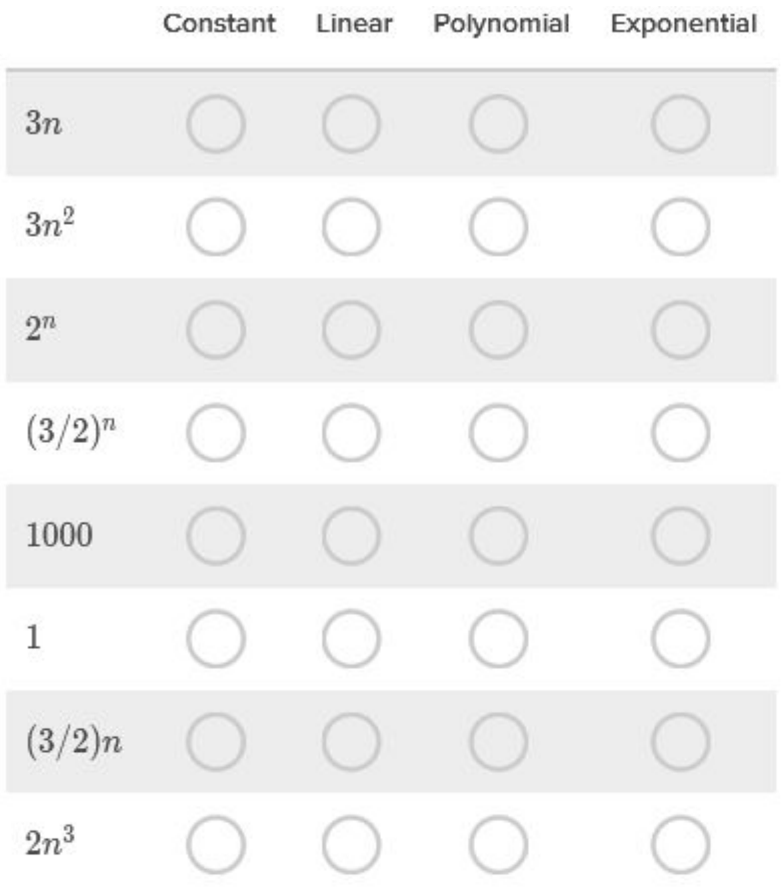
n	log2(n)	n	nlog2(n)	n^2	2^n	n^3
1	0.00	1	0.00	1	2	1
2	1.00	2	2.00	4	4	8
3	1.58	3	4.75	9	8	27
4	2.00	4	8.00	16	16	64
5	2.32	5	11.61	25	32	125
6	2.58	6	15.51	36	64	216
7	2.81	7	19.65	49	128	343
8	3.00	8	24.00	64	256	512
9	3.17	9	28.53	81	512	729
10	3.32	10	33.22	100	1024	1000



Order of growth for varying input size of n

Quiz

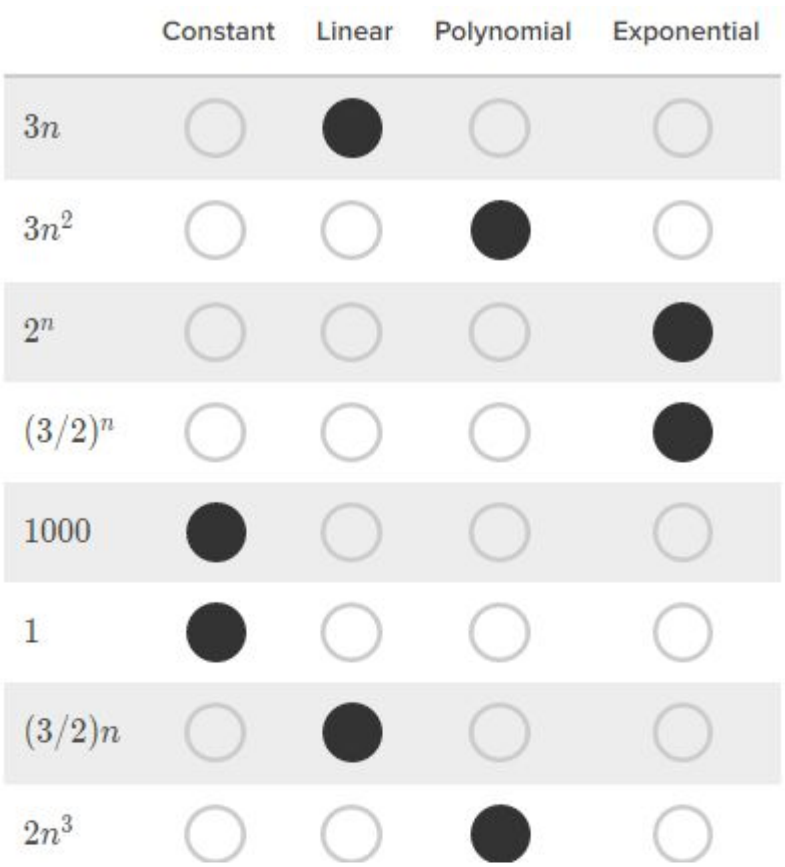
- Which kind of growth best characterizes each of these functions?



	Constant	Linear	Polynomial	Exponential
$3n$	○	○	○	○
$3n^2$	○	○	○	○
2^n	○	○	○	○
$(3/2)^n$	○	○	○	○
1000	○	○	○	○
1	○	○	○	○
$(3/2)n$	○	○	○	○
$2n^3$	○	○	○	○

Answer

- Which kind of growth best characterizes each of these functions?



	Constant	Linear	Polynomial	Exponential
$3n$	○	●	○	○
$3n^2$	○	○	●	○
2^n	○	○	○	●
$(3/2)^n$	○	○	○	●
1000	●	○	○	○
1	●	○	○	○
$(3/2)n$	○	●	○	○
$2n^3$	○	○	●	○

Quiz

- Rank these functions according to their growth, from slowest growing (at the top) to fastest growing (at the bottom).

n^2

2^n

n

n^3

$(3/2)^n$

1

Answer

- Rank these functions according to their growth, from slowest growing (at the top) to fastest growing (at the bottom).

1

n

n^2

n^3

$(3/2)^n$

2^n

Quiz

- Rank these functions according to their growth, from slowest growing to fastest growing.

$6n^3$

$n \log_6 n$

$4n$

$8n^2$

$\log_2 n$

$n \log_2 n$

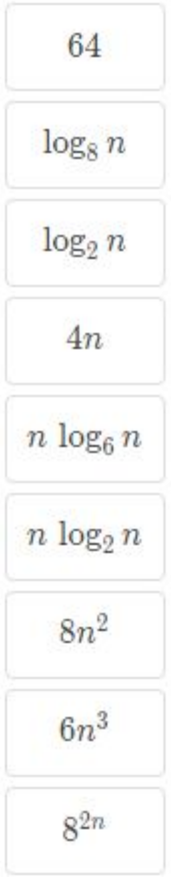
$\log_8 n$

64

8^{2n}

Answer

- Rank these functions according to their growth, from slowest growing to fastest growing.



64

$\log_8 n$

$\log_2 n$

$4n$

$n \log_6 n$

$n \log_2 n$

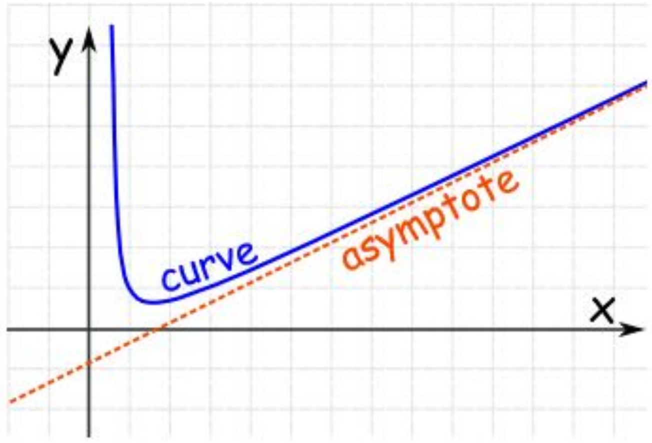
$8n^2$

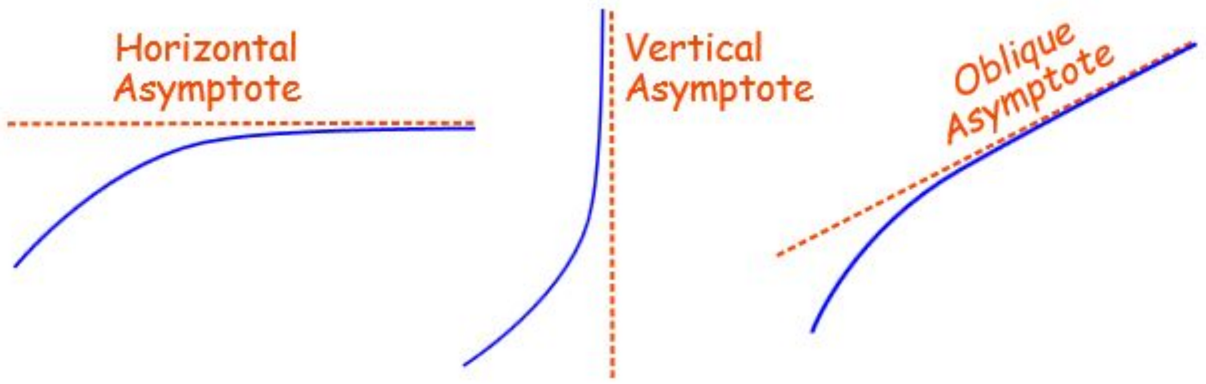
$6n^3$

8^{2n}

Asymptote

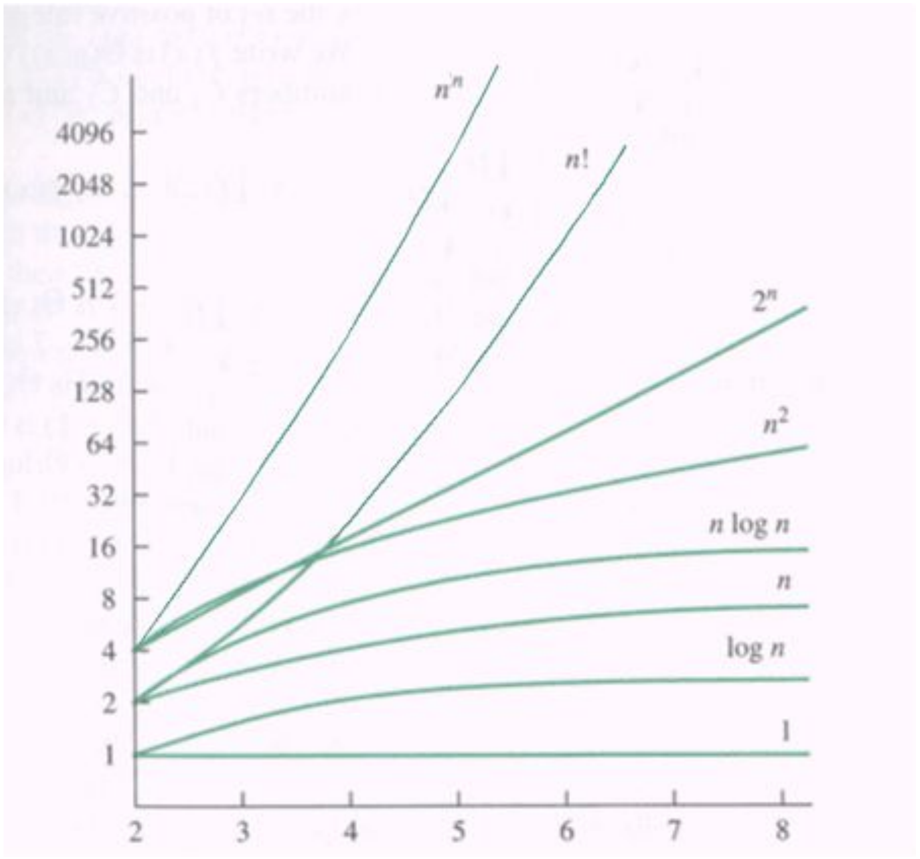
Asymptote: A straight line that continually approaches a given curve but does not meet it at any finite distance





Rate of Growth ordering

$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(2^n) < O(n!)$



Asymptotic notation

- **Asymptotic notation of an algorithm is a mathematical representation of its complexity**
- In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity may be Space Complexity or Time Complexity).

For example, consider the following time complexities of two algorithms...

- **Algorithm 1 : $5n^2 + 2n + 1$**
- **Algorithm 2 : $10n^2 + 8n + 3$**

Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. '**n**' value). In above two time complexities, for larger value of '**n**' the term in algorithm 1 '**$2n + 1$**' has least significance than the term '**$5n^2$**', and the term in algorithm 2 '**$8n + 3$**' has least significance than the term '**$10n^2$**'.

Here for larger value of '**n**' the value of most significant terms (**$5n^2$** and **$10n^2$**) is very larger than the value of least significant terms (**$2n + 1$** and **$8n + 3$**). So for larger value of '**n**' we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

By **dropping the less significant terms** and the **constant coefficients**, we can focus on the important part of an algorithm's running time—its **rate of growth**—without getting mired in details that complicate our understanding. When we drop the constant coefficients and the less significant terms, we use **asymptotic notation**.

We'll see THREE types of Asymptotic Notations:

Big - Oh (O) UPPER BOUNDING function

Big - Omega (Ω) LOWER BOUNDING function

Big - Theta (Θ) ORDER or TIGHT BOUNDING function

Analysis of Linear Search

```
Algorithm SequentialSearch(A[0..n-1,K)
i=0
While i<n and A[i]!=K do
     { i=i+1 }
If i<n
   return i
else
   return -1
```

Question:
If the Key element is in the first position of the Array then
How many times the operation i=i+1 will be executed ?

If the Key element is in the last position of the Array then
How many times the operation i=i+1 will be executed ?

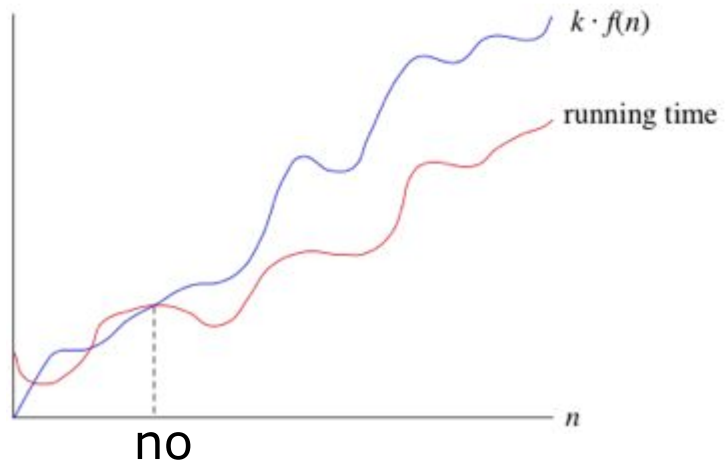
Analysis of Linear Search

```
Algorithm SequentialSearc(A[0..n-1,K)
i=0
While i<n and A[i]!=K do
     { i=i+1 }
If i<n
   return i
else
   return -1
```

Answer:
Find, if the Key element is in the first position of the Array then
How many times the operation i=i+1 will be executed ?
Find, if the Key element is in the last position of the Array then
How many times the operation i=i+1 will be executed ?
Find the total lower bound and upper bound (Best and Worst case)
Running time ? $T_{lower}(n)=1$ $T_{upper}(n)=n$

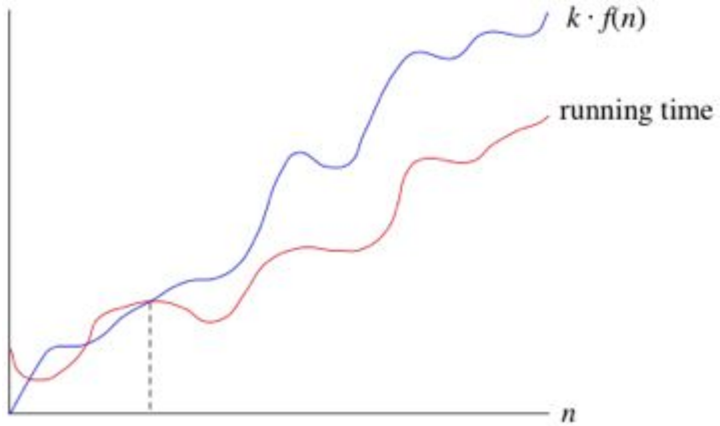
Summarizing Big-O, Big-Omega, Big-theta

Big-O Upper Bound



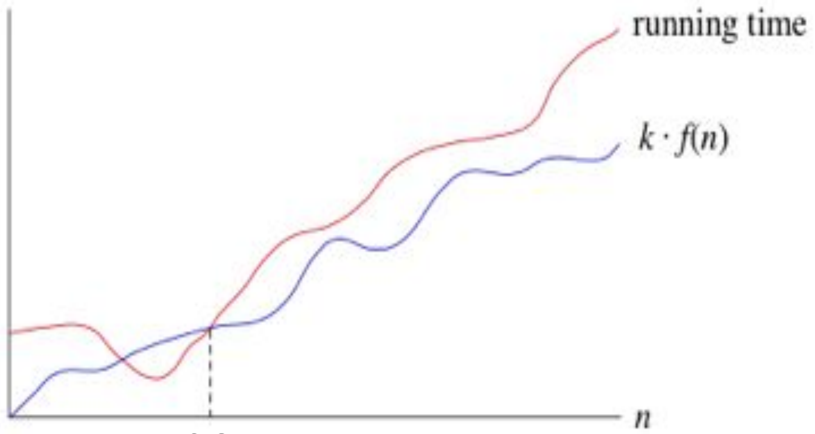
Summarizing Big-O, Big-Omega, Big-theta

Big-O Upper Bound



no

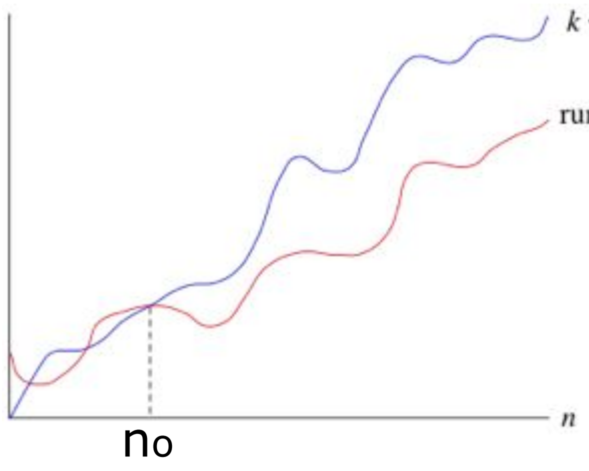
Big-Omega Lower Bound



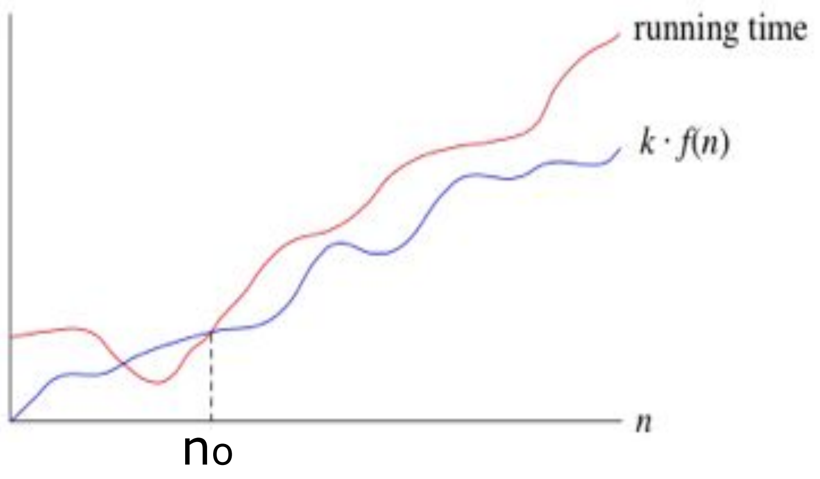
no

Summarizing Big-O, Big-Omega, Big-theta

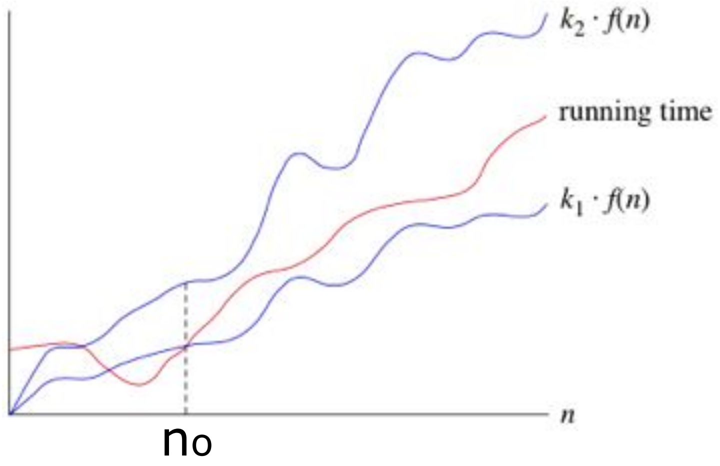
Big-O Upper Bound



Big-Omega Lower Bound



Big-theta
Tight bound or Order bound



Question

Go through the following pseudocode

```
containsZero(arr, n){ #assume normal array of length n
  for i=1 to n {
      if arr[i] == 0  return true
    }
  return false
```

What’s the lower bound or best case? Well, if the array has 0 as the first value, it will take how much time ?

What’s the worst case? If the array doesn’t contain 0, it will take what time.

Question

Go through the following pseudocode

```
containsZero(arr, n){ #assume normal array of length n
  for i=1 to n {
      if arr[i] == 0  return true
    }
  return false
```

What’s the lower bound or best case? Well, if the array we give it has 0 as the first value, it will take what time ?

- Constant time: Ω (1)

What’s the worst case? If the array doesn’t contain 0, it will take what time

- It will iterate through the whole array: O(n)

Question

Go through the following pseudocode

```
printNums(arr,n){
for i=1 to n {
  print(arr[i]);
}
```

Can you think of a best case and worst case??

Question

Go through the following pseudocode

```
printNums(arr,n){
for i=1 to n {
  print(arr[i]);
}
```

Can you think of a best case and worst case??

We can’t! No matter what array we give it, we have to iterate through every value in the array. So the function will take AT LEAST n time (Ω(n)), but we also know it won’t take any longer than n time (O(n)). What does this mean? Our function will take exactly n time i.e., **Θ(n)**

Asymptotic notation

To compare and rank orders of growth or rate of growth of the algorithms, Computer Scientists use three notations:

- ☐ **Big - Oh (O) UPPER BOUNDING function**
- ☐ **Big - Omega (Ω) LOWER BOUNDING function**
- ☐ **Big - Theta (Θ) ORDER BOUNDING function**

Big - Oh (O) UPPER BOUNDING function: Informal Intr.

- Let us consider t(n) and g(n) are non-negative functions (or expressions) which take non-negative arguments.
- **O(g(n)) is set of all functions with a smaller or same order of growth g(n)** (to within a constant multiple, as n goes to infinity).

Ex: $n \in O(n^2)$ for all $n>=1$

Informal Introduction

- Let us consider t(n) and g(n) are non-negative functions (or expressions) which take non-negative arguments.
- **O(g(n)) is set of all functions with a smaller or same order of growth g(n)** (to within a constant multiple, as goes to infinity).

Ex: $n \in O(n^2)$ for all $n>=1$

$n \in O(n^2)\ \forall\ n>=1$	
$n <= n^2$	
n	n^2
1	1
2	4
3	9

Informal Introduction

- Let us consider t(n) and g(n) are non-negative functions (or expressions) which take non-negative arguments.
- **O(g(n)) is set of all functions with a smaller or same order of growth g(n)** (to within a constant multiple, as goes to infinity).
- Ex: $100n+5 \in O(n^2)$ for all $n>=1000$

Big-Oh Upper Bounding:Informal Intr.

- Let us consider t(n) and g(n) are non-negative functions (or expressions) which take non-negative arguments.
- **O(g(n)) is set of all functions with a smaller or same order of growth g(n)** (to within a constant multiple, as goes to infinity).
- Ex: $100n+5 \in O(n^2)$ for all $n>=1000$

$100n+5 \in O(n^2)$ $\forall\ n>=10^3$		
$100n+5 <= n^2$		
n	100n+5	n^2
10^2	10^4+5	10^4
10^3	**10^5+5**	**10^6**
10^4	10^6+5	10^8

Question

- Check whether the statement $\frac{1}{2}n(n-1) \in O(n^2)$ is true

Answer

Check whether the statement

$\frac{1}{2}n(n-1)$ ∈ $O(n^2)$ is true

True for all n >=1

n	(1/2)n(n-1)	n^2
1	0	1
2	1	4
3	3	9
4	6	16
5	10	25

Question

Check which of the following statement is true

- ☐ $n^3 \in O(n^2)$
- ☐ $n^3 \notin O(n^2)$

Answer

Check which of the following statement is true

- $n^3 \in O(n^2)$ False
- $n^3 \notin O(n^2)$ True

Question

Check whether the following statement is true

- $n^4 + n + 1 \notin O(n^2)$

Answer

Check whether the following statement is true

- $n^4 + n + 1 \notin O(n^2)$

True

Formal Definition

Big - Oh (O) UPPER BOUNDING function

Big - Oh (O) UPPER BOUNDING function

- Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity. That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Formal Definition:

- A function f(n) is said to be in O(g(n)) , denoted

$$\mathbf{f(n) \in O(g(n))} \quad \text{(or } f(n) = O(g(n))\text{)},$$

if f(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$f(n) <= cg(n) \text{ for all } n>=n_0$$

(f(n) is less than or equal to cg(n) for all values of n greater than or equal to n_0)

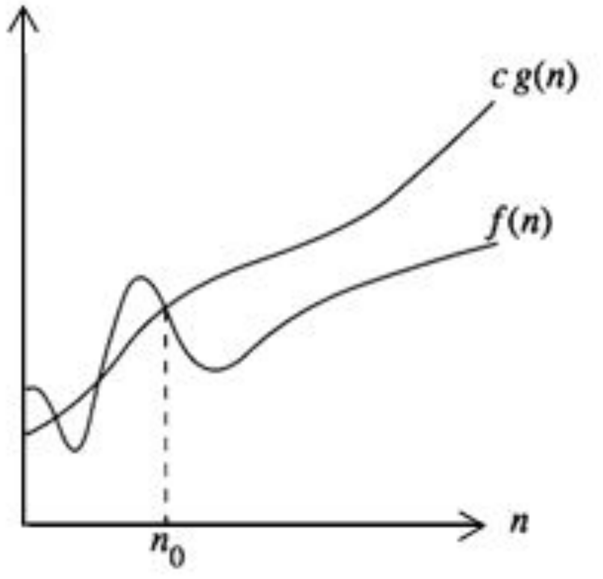
Big - Oh (O) UPPER BOUNDING function

- Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity. That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Formal Definition

- A function f(n) is said to be in O(g(n)) , denoted **f(n) ∊ O(g(n)) (or f(n) = O(g(n)))**, if f(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

  $f(n) <= cg(n)$ for all $n>=n_0$



n is Size of program's input.

f(n) Any real world function. Example: - Running time of a machine.

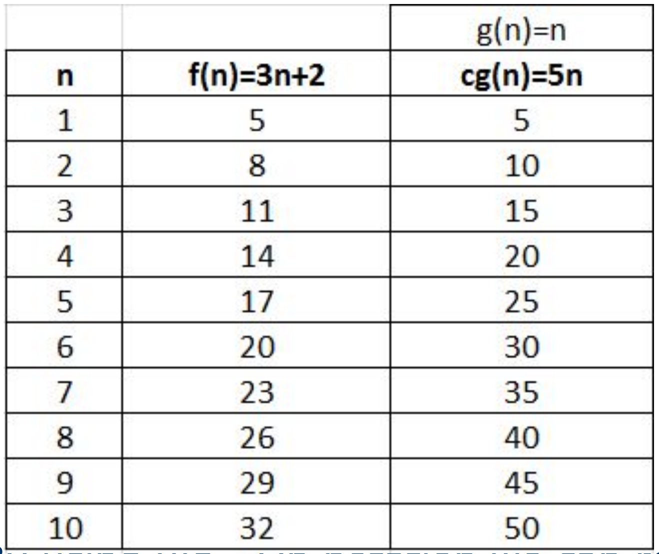
g(n) Another function that we want to use as an upper-bound. Not a real world function but preferably simple.

Example

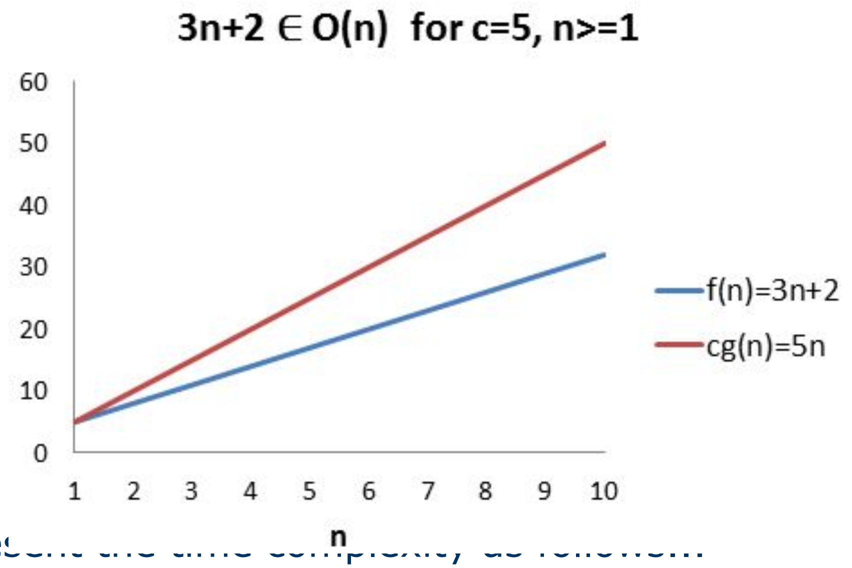
- Consider the following f(n) and g(n)...
  f(n) = 3n + 2
  g(n) = n
  If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= Cg(n)** for all values of **C > 0** and **n_0>= 1**

Example

- Consider the following t(n) and g(n)...
  f(n) = 3n + 2
  g(n) = n
  If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= Cg(n)** for all values of **C > 0** and **n_0>= 1**
- f(n) <= C g(n) ⇒ 3n + 2 <= Cn
  Above condition is always TRUE for all values of **C = 5** and **n >= 1**.



		g(n)=n
n	**f(n)=3n+2**	**cg(n)=5n**
1	5	5
2	8	10
3	11	15
4	14	20
5	17	25
6	20	30
7	23	35
8	26	40
9	29	45
10	32	50



By using Big - Oh notation we can represent the time complexity as follows...
3n + 2 ∈ O(n) or 3n + 2 = O(n)

Example

Consider the following t(n) and g(n)...

$t(n) = 6*2^n + n^2$

$g(n) = 2^n$

Represent $6*2^n + n^2 \in O(2^n)$

$C = ??$, $n_0 = ??$

i.e., $6*2^n + n^2 <= C(2^n)$ for all $n >= n_0$

Example

Consider the following t(n) and g(n)...

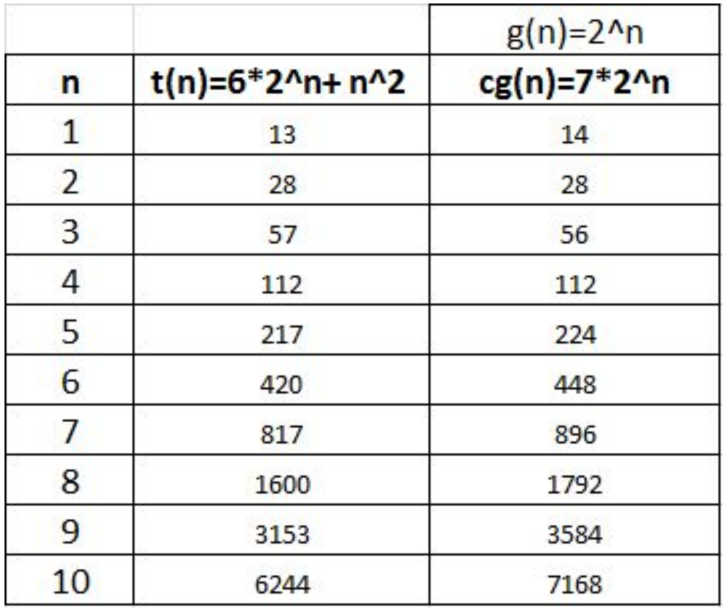
$t(n) = 6*2^n + n^2$

$g(n)=2^n$

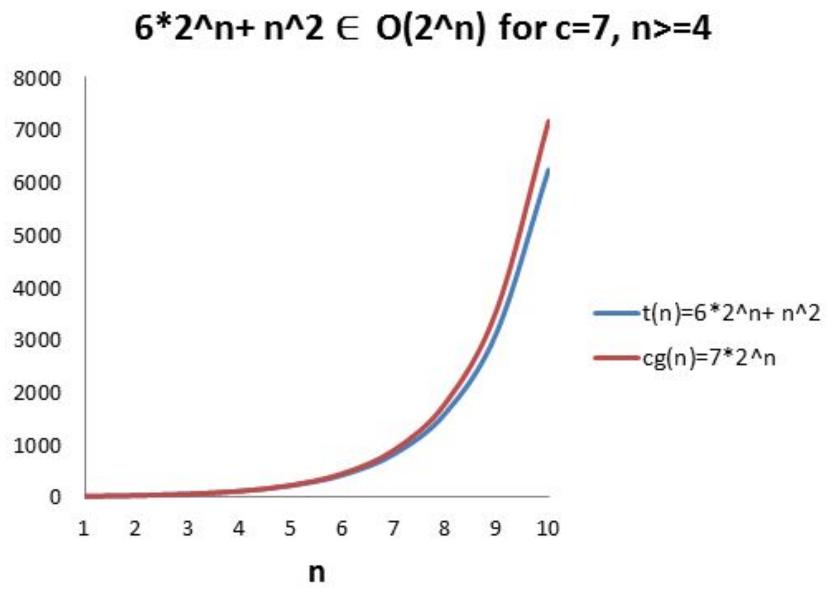
Represent $6*2^n + n^2 \in O(2^n)$

$C=7, n_0=4$

i.e., $6*2^n + n^2 <= 7(2^n)$ for all $n >=4$



		g(n)=2^n
n	t(n)=6*2^n+ n^2	cg(n)=7*2^n
1	13	14
2	28	28
3	57	56
4	112	112
5	217	224
6	420	448
7	817	896
8	1600	1792
9	3153	3584
10	6244	7168



Question

Find the values for **c** and **n** to show that the assertion $3n^3 + 2n^2 \in O(n^3)$ is true

Question

Find the values for **c** and **n** to show that the assertion $3n^3 + 2n^2 \in O(n^3)$ is true

Answer:

- $C=5$, $n_0=1$
- $3n^3 + 2n^2 <= 5(n^3)$ for all $n >=1$

Question

- Is the following assertion True

$3^n \notin O(2^n)$

Question

- Is the following assertion True

$3^n \notin O(2^n)$

Answer: Yes

Question

Prove that $n^3 + n^2 + n \in O(n^3)$

Question

Prove that $n^3 + n^2 + n \in O(n^3)$,

- ***Sol.*** For C=3, and n_0=1,

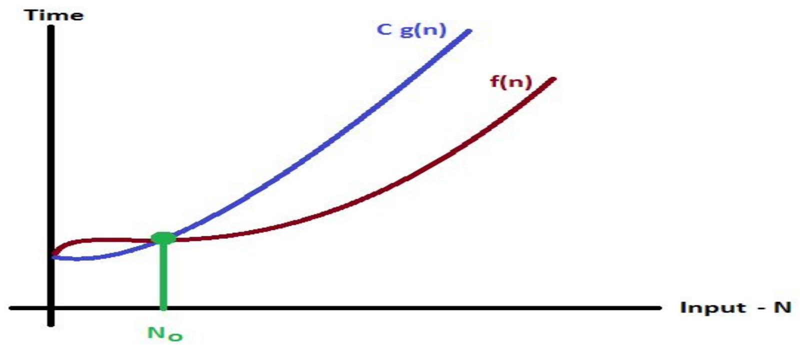
Big - Oh Notation (O)

- Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.
  That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.
  Big - Oh Notation can be defined as follows...

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).

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

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



- In above graph a [illegible] er than f(n) which indicates the algorithm's upper bound.

Big-O

- Big-O, commonly written as **O**, is an Asymptotic Notation for the worst case, or ceiling of growth for a given function. It provides us with an ***asymptotic upper bound*** for the growth rate of runtime of an algorithm. Say f(n) is your algorithm runtime, and g(n) is an arbitrary time complexity you are trying to relate to your algorithm. f(n) is O(g(n)), if for some real constants c ($c > 0$) and n_0, $f(n) <= c\ g(n)$ for every input size n ($n > n_0$).

Example 1

- f(n) **=** 3log n **+** 100 g(n) **=** log n
- Is f(n) =O(g(n))? Is 3 log n + 100=O(log n)? Let's look to the definition of Big-O.
- **3**log n **+** 100 **<=** c ***** log n
- Is there some pair of constants c, n_0 that satisfies this for all $n > n_0$?
- **3**log n **+** 100 **<=** 150 ***** log n, n **>** 2 (undefined at n **=** 1)
- Yes! The definition of Big-O has been met therefore f(n) is O(g(n)).

Big-O

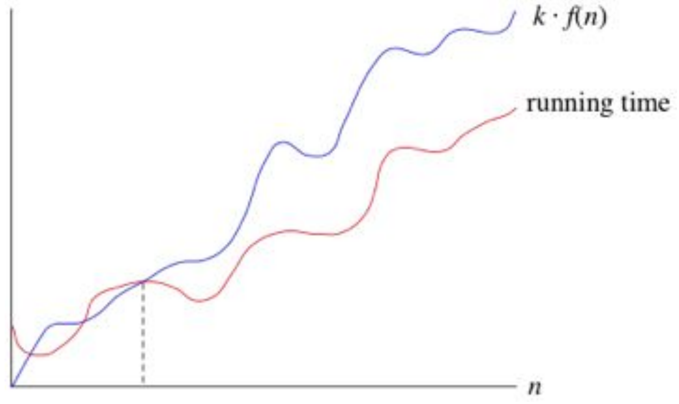
- Big-O, commonly written as **O**, is an Asymptotic Notation for the worst case, or ceiling of growth for a given function. It provides us with an ***asymptotic upper bound*** for the growth rate of runtime of an algorithm. Say f(n) is your algorithm runtime, and g(n) is an arbitrary time complexity you are trying to relate to your algorithm. f(n) is O(g(n)), if for some real constants c (c > 0) and n_0, f(n) <= c g(n) for every input size n (n > n_0).

Example 2

- f(n) **=** 3*****n**^**2 g(n) **=** n
- Is f(n) O(g(n))? Is 3 * n^2 O(n)? Let's look at the definition of Big-O.
- **3 *** n^2 **<=** c ***** n
- Is there some pair of constants c, n_0 that satisfies this for all n > $_0$? No, there isn't. f(n) is NOT O(g(n)).

Summarizing Big-Oh

- It would be convenient to have a form of asymptotic notation that means "the running time grows at most this much, but it could grow more slowly." We use "big-O" notation for just such occasions.
- If a running time is O(f(n)), then for large enough n, the running time is at most *k f*(*n*) for some constant k. Here's how to think of a running time that is O(f(n)):



- We say that the running time is "big-O of f(n)" or just "O of f(n)." We use big-O notation for **asymptotic upper bounds**, since it bounds the growth of the running time from above for large enough input sizes.

Formal Definition

Big - Omega (Ω) LOWER BOUNDING function

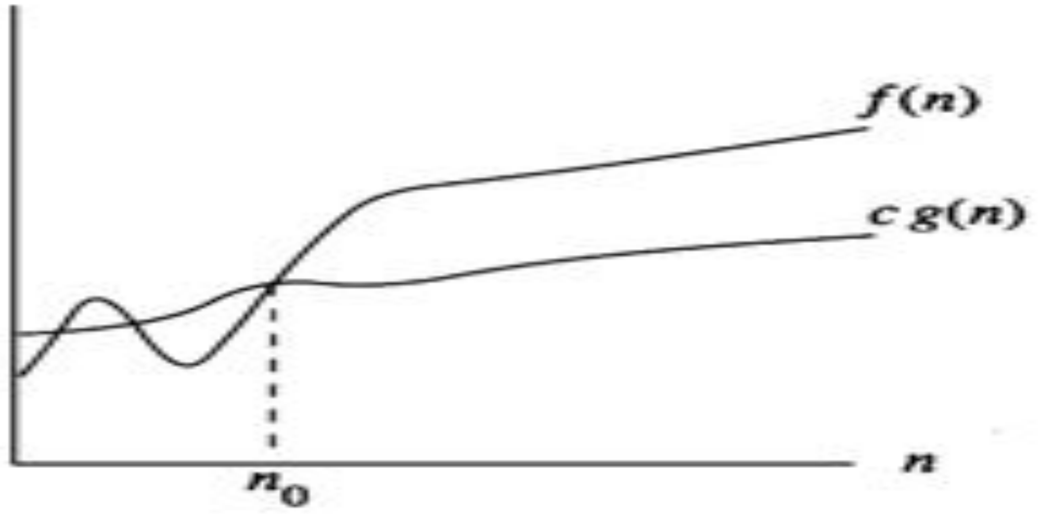
Big - Omega (Ω) LOWER BOUNDING function

- Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity. That means Big - Omega notation always indicates the **minimum time required** by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Formal Definition

- A function f(n) is said to be in **Ω**(g(n)) , denoted f(n) ∈ **Ω**(g(n)), if f(n) is bounded below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$f(n) >= cg(n) \text{ for all } n>=n_0$$



Big - Omega (Ω)

Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

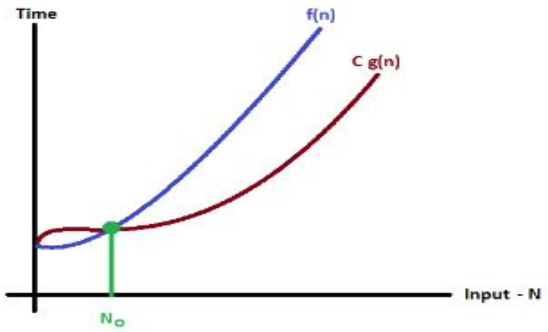
That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If $f(n) >= C \times g(n)$ for all $n >= n_0$, $C > 0$ and $n_0 >= 1$. Then we can represent f(n) as Ω(g(n)).

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

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C x g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

- Consider the following f(n) and g(n)...
  f(n) = 3n + 2
  g(n) = n
  If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**

Example

- Consider the following f(n) and g(n)...
  f(n) = 3n + 2
  g(n) = n
  If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**

- f(n) >= C g(n)
  ⇒3n + 2 >= C n
  Above condition is always TRUE for all values of
  C = 1 and **n >= 1**.

- By using Big - Omega notation we can represent the time complexity as follows...
  3n + 2 = Ω(n)

Example

- Consider the following f(n) and g(n)...
  f(n) = $n^3 + 4n^2$
  g(n) = n^2

If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**

Example

- Consider the following f(n) and g(n)...
  f(n) = $n^3 + 4n^2$
  g(n) = n^2

If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**

- f(n) >= C g(n)
  $\Rightarrow n^3 + 4n^2 >= C\ n^2$
  Above condition is always TRUE for all values of
  C = 1 and **n >= 1**.

- By using Big - Omega notation we can represent the time complexity as follows...
  $n^3 + 4n^2$ **= Ω(**n^2**)**

Big-Oh and Big-Omega

- Big-Oh
- Think of it this way. Suppose you have 10 rupees in your pocket. You go up to your friend and say, "I have an amount of money in my pocket, and I guarantee that **it's no more** than one thousand rupees." Your statement is absolutely true, though not terribly precise.

- Big-Omega
- For example, if you really do have a one thousand rupees in your pocket, you can truthfully say "I have an amount of money in my pocket, and it's **at least** 10 rupees." That is correct, but certainly not very precise.

Formal Definition

Big - Theta (Θ) ORDER BOUNDING function

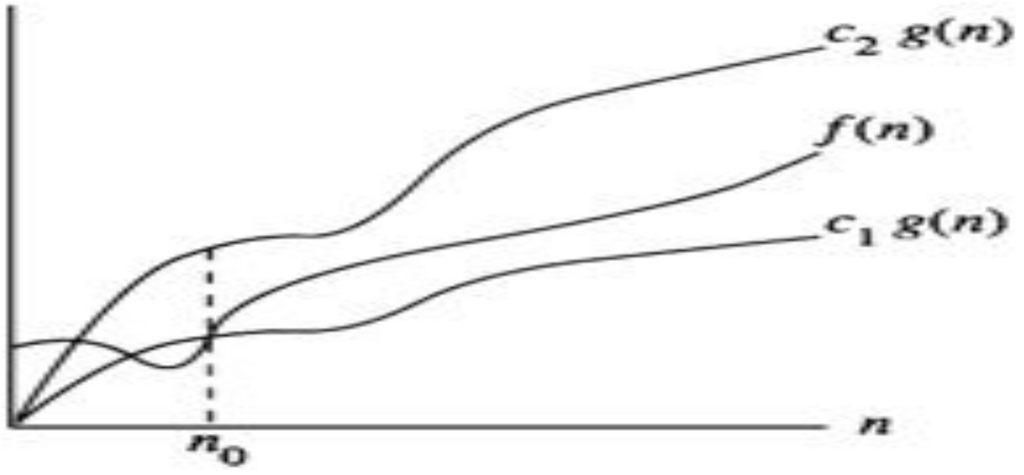
Big - Theta (Θ) ORDER BOUNDING function

- Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity. That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Formal Definition

- A function f(n) is said to be in **Θ**(g(n)) , denoted f(n) ∈ **Θ**(g(n)), if f(n) is bounded above and below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that

$$c_1 g(n) <= f(n) <= c_2 g(n) \text{ for all } n>=n_0$$



Example

- Consider the following f(n) and g(n)...
  $f(n) = 3n + 2$
  $g(n) = n$
  If we want to represent **$f(n)$** as **$\Theta(g(n))$** then it must satisfy **$C_1\, g(n) <= f(n) <= C_2\, g(n)$** for all values of **$C_1, C_2 > 0$** and **$n_0 >= 1$**

Example

- Consider the following f(n) and g(n)...
  f(n) = 3n + 2
  g(n) = n
  If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **C_2 g(n) <= f(n) <= C_1 g(n)** for all values of **C_1, C_2 > 0** and **n_0>= 1**
- C_1 g(n) <= f(n) <= C_2 g(n) ⇒

  C_1 n <= 3n + 2 <= C_2 n

Above condition is always TRUE for all values of **C_1 = 1, C_2 = 5** and **n >= 1**.
By using Big - Theta notation we can represent the time complexity as follows...
3n + 2 = Θ(n)

Example

- Consider the following f(n) and g(n)...
  $f(n) = 10n^3+5$
  $g(n) = n^3$

If we want to represent **$f(n)$** as **$\Theta(g(n))$** then it must satisfy **$C_2\ g(n) <= f(n) <= C_1\ g(n)$** for all values of **$C_1, C_2 > 0$** and **$n_0 >= 1$**

Example

- Consider the following f(n) and g(n)...
  $f(n) = 10n^3+5$
  $g(n) = n^3$

If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **$C_1\ g(n) <= f(n) <= C_2\ g(n)$** for all values of **$C_1, C_2 > 0$** and **$n_0 >= 1$**

- $C_1\ g(n) <= f(n) <= C_2\ g(n) \Rightarrow$
  $C_1\ \mathbf{n^3} <= \mathbf{10n^3+5} <= C_2\ \mathbf{n^3}$

  Above condition is always TRUE for all values of
  $C_1 = 10, C_2 = 11$ and **$n >= 2$**.

By using Big - Theta notation we can represent the time complexity as follows...
$10n^3+5 = \Theta(n^3)$

Summarizing:
Formal Definitions of Asymptotic Notations

Big - Oh (O) UPPER BOUNDING function

Big - Omega (Ω) LOWER BOUNDING function

Big - Theta (Θ) ORDER BOUNDING function

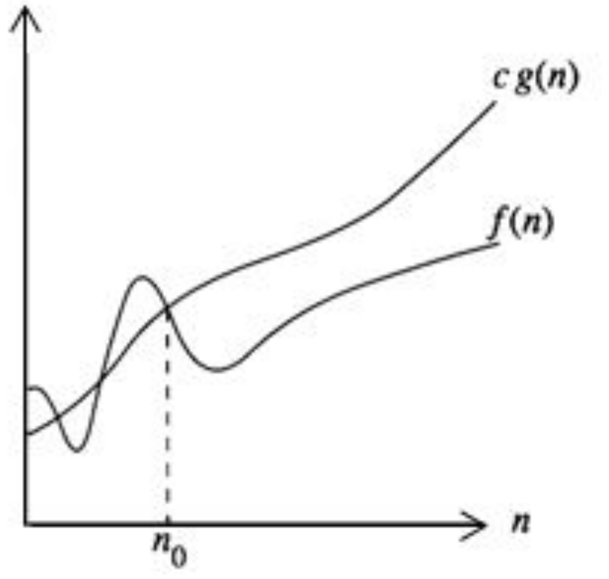
Big - Oh (O) UPPER BOUNDING function

- Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity. That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Formal Definition

- A function f(n) is said to be in O(g(n)) , denoted **f(n) ∈ O(g(n)) (or f(n) = O(g(n)))**, if f(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

  $f(n) <= cg(n)$ for all $n>=n_0$



n is Size of program's input.

t(n) Any real world function. Example: - Running time of a machine.

g(n) Another function that we want to use as an upper-bound. Not a real world function but preferably simple.

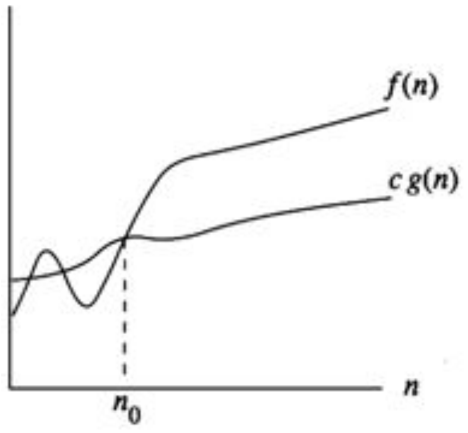
Big - Omega (Ω) LOWER BOUNDING function

- Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity. That means Big - Omega notation always indicates the **minimum time required** by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Formal Definition

- A function f(n) is said to be in **Ω**(g(n)) , denoted f(n) ∈ **Ω**(g(n)), if f(n) is bounded below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$f(n) >= cg(n) \text{ for all } n>=n_0$$



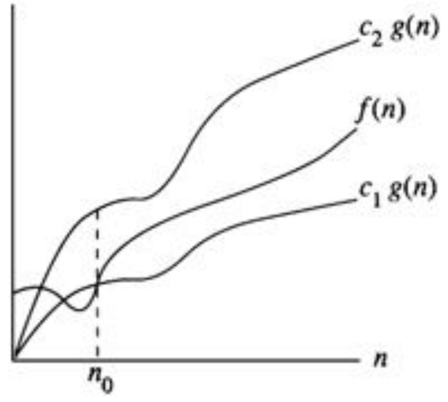
Big - Theta (Θ) ORDER BOUNDING function

- Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity. That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Formal Definition

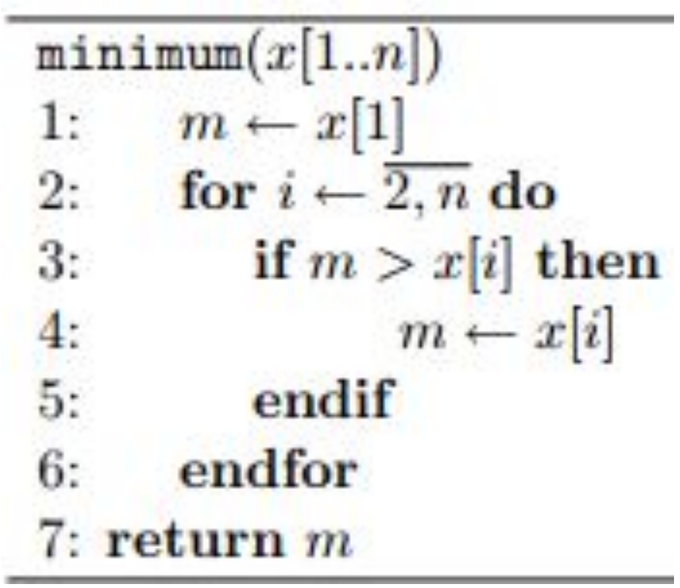
- A function f(n) is said to be in **Θ**(g(n)) , denoted f(n) ∈ **Θ**(g(n)), if f(n) is bounded above and below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that

$$c_1 g(n) <= f(n) <= c_2 g(n) \text{ for all } n>=n_0$$



Question

Let us consider the problem of finding the minimum of an array x[1..n]. The input size is n and the corresponding algorithm can be described as follows:

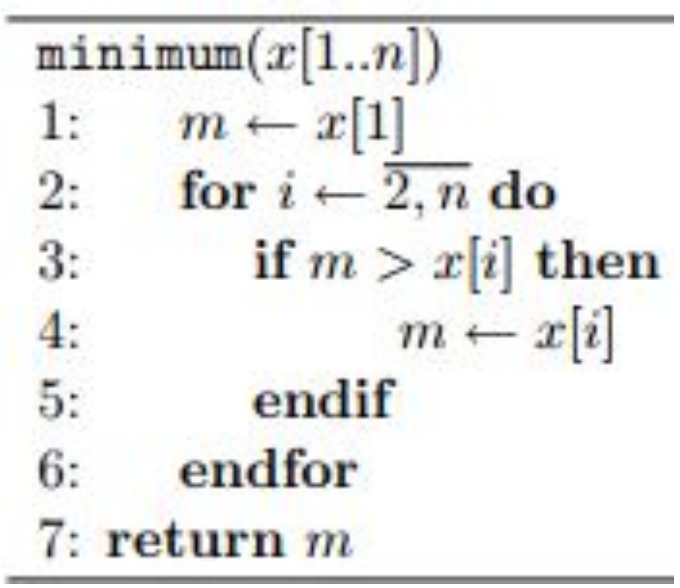


```
minimum($x[1..n]$)
1:    $m \leftarrow x[1]$
2:    for $i \leftarrow \overline{2, n}$ do
3:        if $m > x[i]$ then
4:            $m \leftarrow x[i]$
5:        endif
6:    endfor
7: return $m$
```

In the given algorithm above, how many times the fourth operation (m=x[i]) will be executed if the given array (x) is having elements in **ascending order** ?

Answer

Let us consider the problem of finding the minimum of an array x[1..n]. The input size is n and the corresponding algorithm can be described as follows:



```
minimum($x[1..n]$)
1:    $m \leftarrow x[1]$
2:    for $i \leftarrow \overline{2, n}$ do
3:        if $m > x[i]$ then
4:            $m \leftarrow x[i]$
5:        endif
6:    endfor
7: return $m$
```

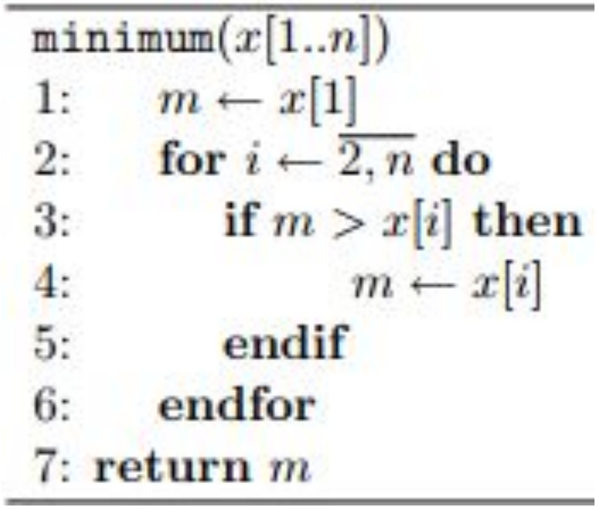
It will not get executed

Best Case (Lower Bound)
T(n) = 0

In the given algorithm above, how many times the fourth operation (m=x[i]) will be executed if the given array (x) is having elements in **ascending order** ?

Question

Let us consider the problem of finding the minimum of an array x[1..n]. The input size is n and the corresponding algorithm can be described as follows:

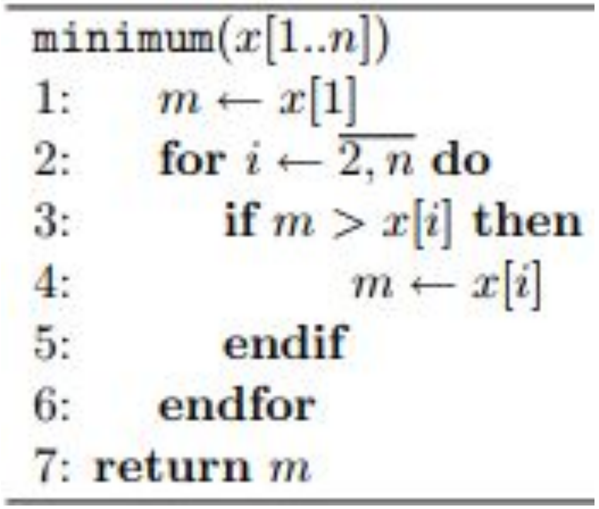


```
minimum($x[1..n]$)
1:    $m \leftarrow x[1]$
2:    for $i \leftarrow \overline{2,n}$ do
3:        if $m > x[i]$ then
4:            $m \leftarrow x[i]$
5:        endif
6:    endfor
7: return $m$
```

In the given algorithm above, how many times the fourth operation (m=x[i]) will be executed if the given array (x) is having elements in **Descending order** ?

Answer

Let us consider the problem of finding the minimum of an array x[1..n]. The input size is n and the corresponding algorithm can be described as follows:



```
minimum($x[1..n]$)
1:    $m \leftarrow x[1]$
2:    for $i \leftarrow \overline{2, n}$ do
3:        if $m > x[i]$ then
4:            $m \leftarrow x[i]$
5:        endif
6:    endfor
7: return $m$
```

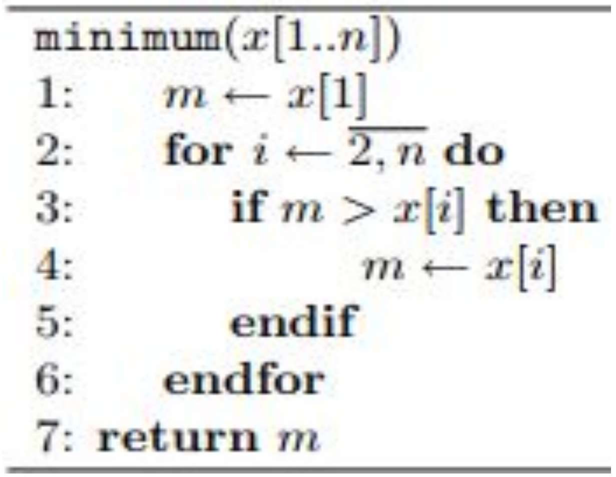
It will get executed (n-1) times

Worst Case (Upper bound)

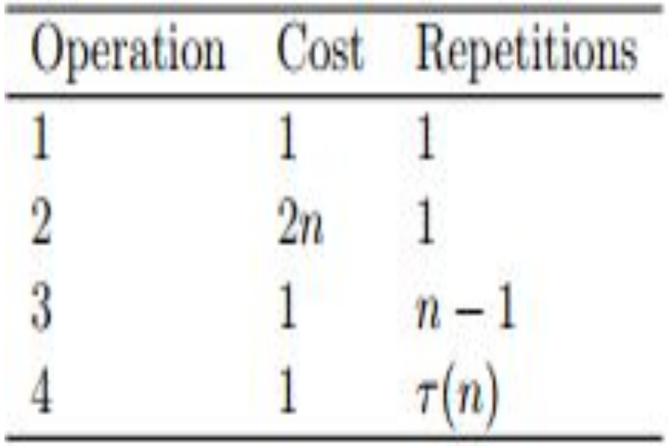
T(n) = (n-1)

In the given algorithm above, how many times the fourth operation (m=x[i]) will be executed if the given array (x) is having elements in **Descending order** ?

Best and Worst case



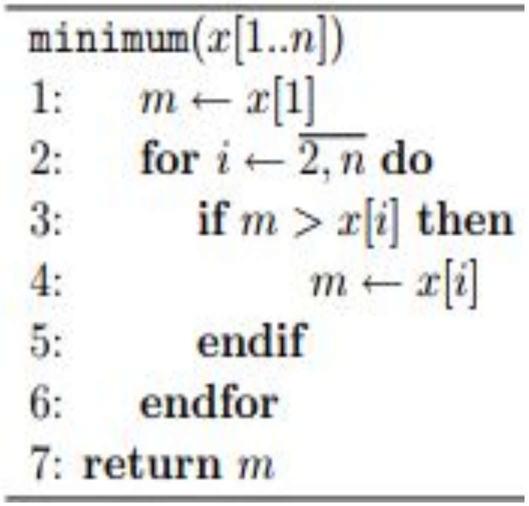
```
minimum($x[1..n]$)
1:    $m \leftarrow x[1]$
2:    for $i \leftarrow \overline{2, n}$ do
3:        if $m > x[i]$ then
4:            $m \leftarrow x[i]$
5:        endif
6:    endfor
7: return $m$
```



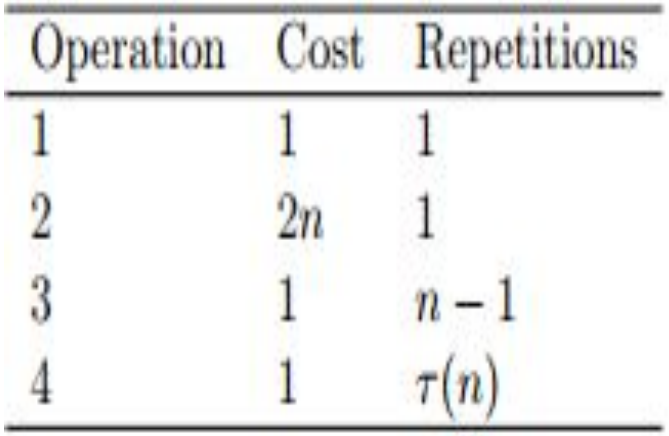
Operation	Cost	Repetitions
1	1	1
2	$2n$	1
3	1	$n-1$
4	1	$\tau(n)$

Unlike in previous examples we cannot find a general expression for the running time. This happens because the running time of the fourth operation does not depend only on the input size but also on the properties of the array (mainly on the position where the minimal value appears for the first time).

Best and Worst case



```
minimum($x[1..n]$)
1: $m \leftarrow x[1]$
2: for $i \leftarrow \overline{2,n}$ do
3:     if $m > x[i]$ then
4:         $m \leftarrow x[i]$
5:     endif
6: endfor
7: return $m$
```



Operation	Cost	Repetitions
1	1	1
2	$2n$	1
3	1	$n-1$
4	1	$\tau(n)$

If the minimum is on the first position then the assignment 4 is not at all executed, τ (n) = 0. This is the **best case** which could appear.

If, on the other hand, the array is decreasingly sorted the assignment 4 is executed at each iteration. Thus τ(n) = n − 1. This is the **worst case**.

Taking into consideration the **best** and the **worst case** we can establish a **lower** and an **upper bound** for the running time:

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

$3n \leq T(n) \leq 4n-1$. It is easy to see that both bounds depend linearly on the input size.

Example

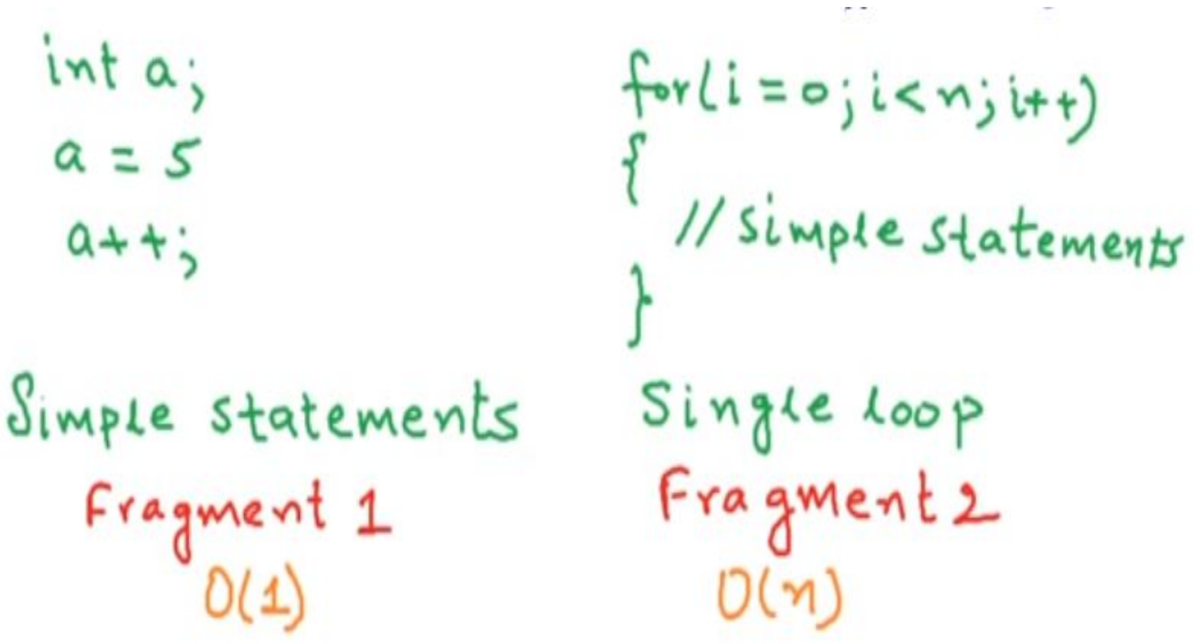
```
int a;
a = 5
a++;
```

Simple statements

Fragment 1

O(1)

Example



```
int a;
a = 5
a++;
```

Simple statements

Fragment 1

O(1)

```
for(i=0;i<n;i++)
{
  //simple statements
}
```

Single loop

Fragment 2

O(n)

Example

```
int a;
a = 5
a++;
```

Simple statements

Fragment 1

$O(1)$

```
for(i=0;i<n;i++)
{
   // Simple Statements
}
```

Single loop

Fragment 2

$O(n)$

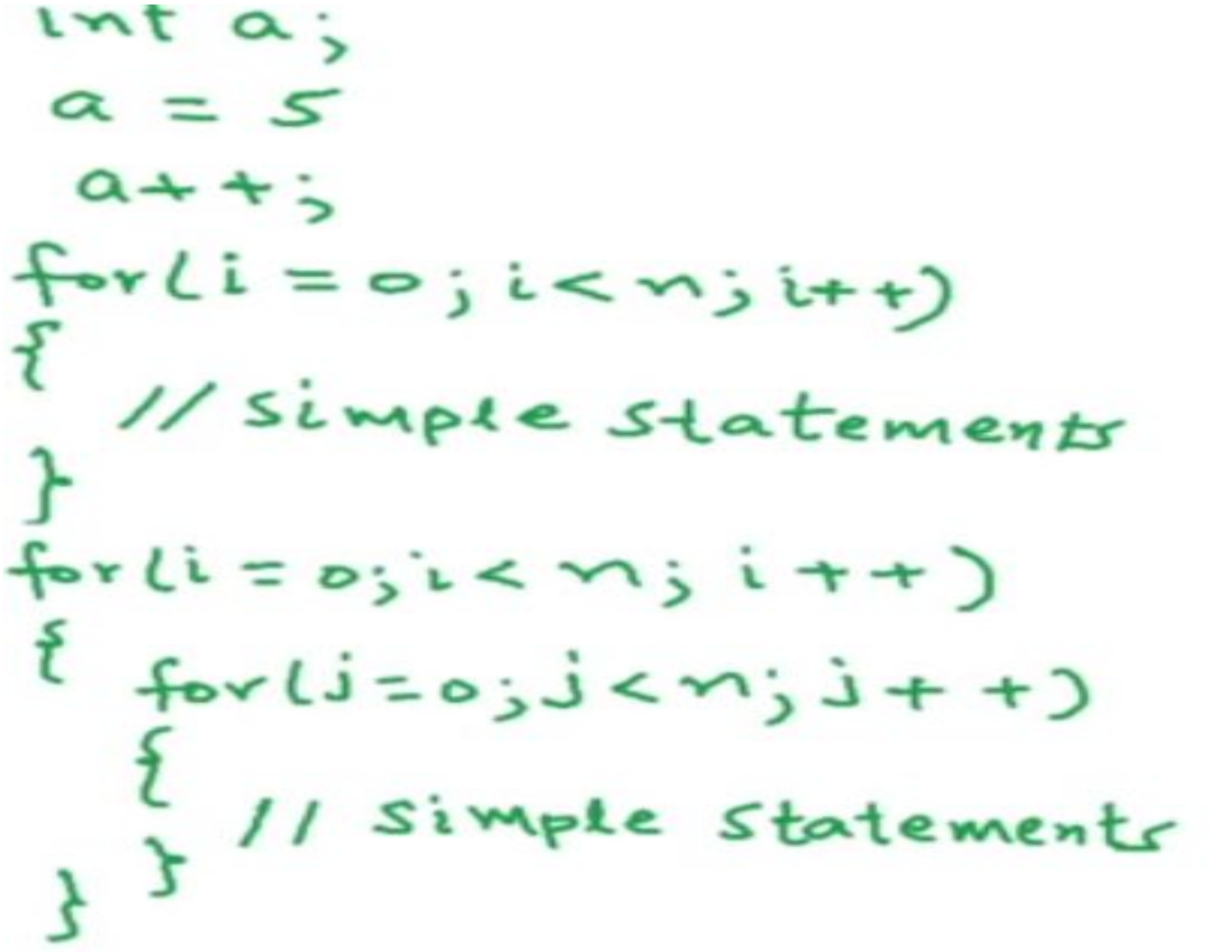
```
for(i=0;i<n; i++)
{
  for(j=0;j<n;j++)
  {
     // Simple Statement
  }
}
```

Nested Loop

Fragment 3

$O(n^2)$

Question: What is the running time of this code ?



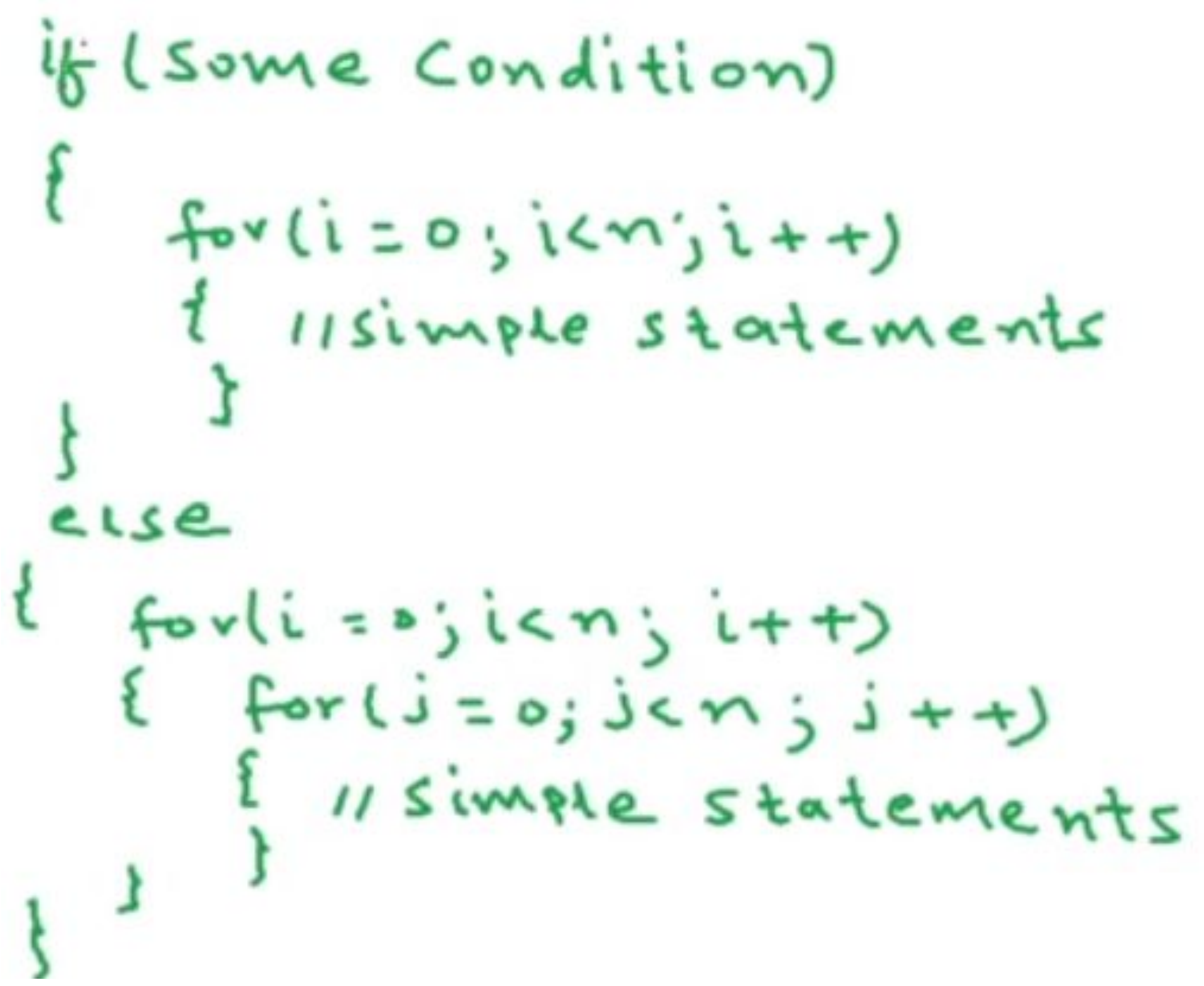
```
int a;
 a = 5
 a++;
for(i=o;i<n;i++)
{
   // simple statements
}
for(i=o;i<n; i++)
 {  for(j=o;j<n;j++)
     {
        // Simple Statements
     }
 }
```

Answer

$$T(n) = O(1) + O(n) + O(n^2) = O(n^2)$$

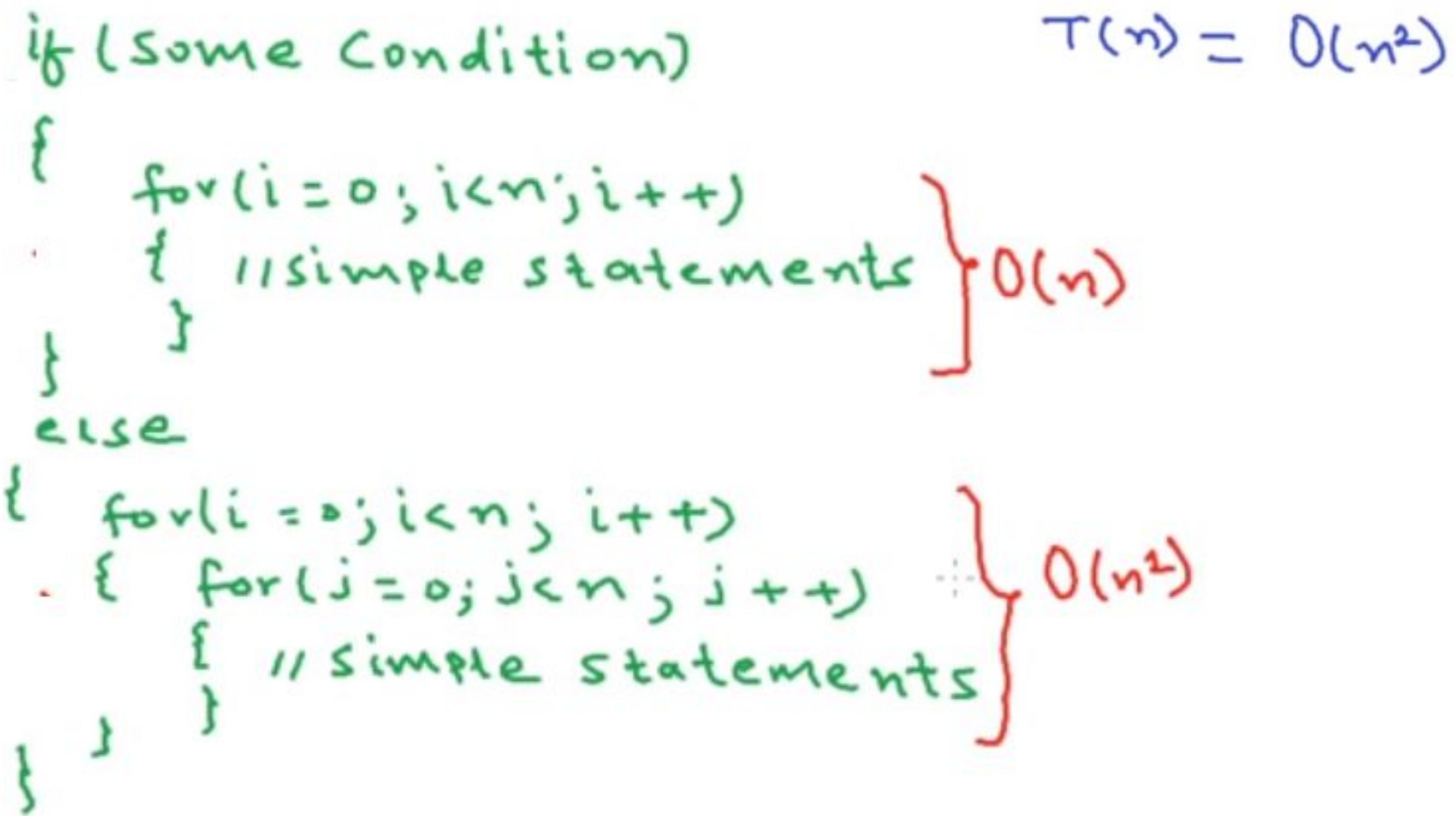
```
int a;      }
a = 5       }  O(1)
a++;        }

for(i=0;i<n;i++)              }
{                             }
  // simple statements        }  O(n)
}                             }

for(i=0;i<n;i++)              }
{                             }
  for(j=0;j<n;j++)            }
  {                           }  O(n^2)
    // simple statements      }
  }                           }
}                             }
```

Question: What is the running time of this code ?



```
if (some Condition)
{
    for(i=0;i<n;i++)
    { //simple statements
    }
}
else
{   for(i=0;i<n; i++)
    {   for(j=0;j<n; j++)
        { // simple statements
        }
    }
}
```

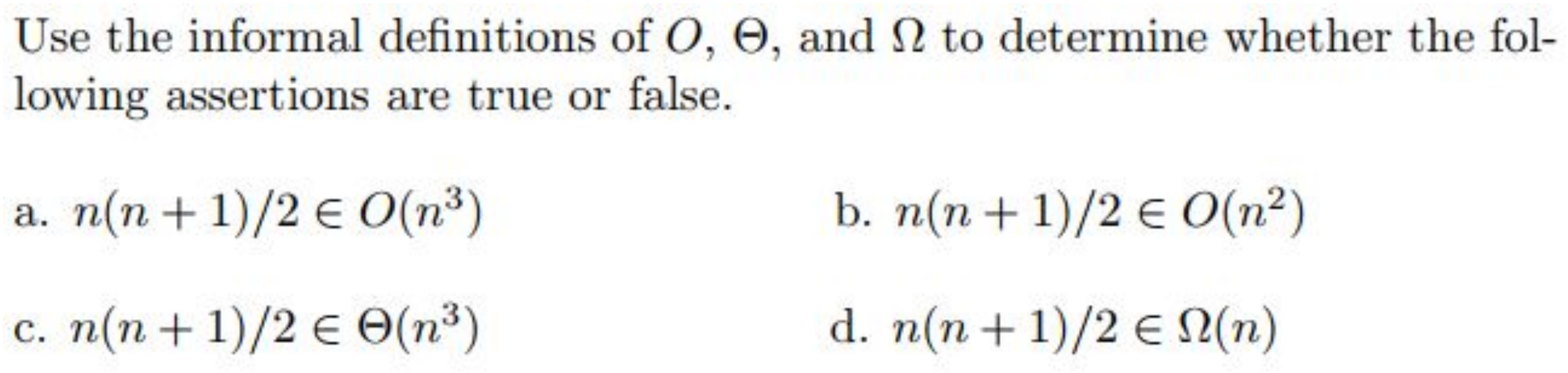
Answer



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

```
if (some condition)
{
    for(i=0; i<n; i++)
    {   //simple statements          } O(n)
    }
}
else
{
  for(i=0; i<n; i++)
  {   for(j=0; j<n; j++)             } O(n^2)
      {  //simple statements
      }
  }
}
```

Question



Use the informal definitions of O, Θ, and Ω to determine whether the following assertions are true or false.

a. $n(n+1)/2 \in O(n^3)$

b. $n(n+1)/2 \in O(n^2)$

c. $n(n+1)/2 \in \Theta(n^3)$

d. $n(n+1)/2 \in \Omega(n)$

Question

Use the informal definitions of O, Θ, and Ω to determine whether the following assertions are true or false.

a. $n(n+1)/2 \in O(n^3)$ b. $n(n+1)/2 \in O(n^2)$

c. $n(n+1)/2 \in \Theta(n^3)$ d. $n(n+1)/2 \in \Omega(n)$

Answer

$n(n+1)/2 \approx n^2/2$ is quadratic. Therefore

a. $n(n+1)/2 \in O(n^3)$ is true. b. $n(n+1)/2 \in O(n^2)$ is true.

c. $n(n+1)/2 \in \Theta(n^3)$ is false. d. $n(n+1)/2 \in \Omega(n)$ is true.

Question

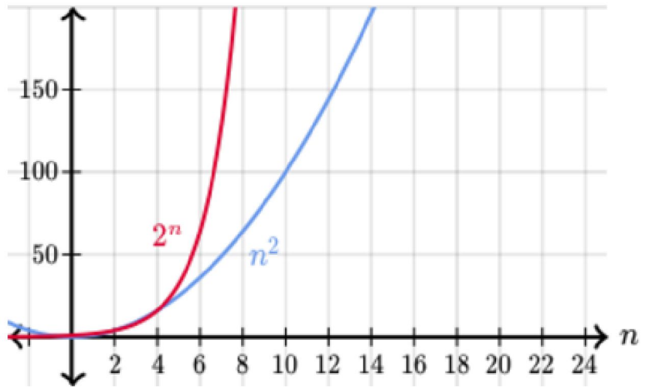
- For the functions, n^k and c^nc, what is the asymptotic relationship between these functions? Assume that k >= 1 and c > 1 are constants.
- n^k is O(c^n)
- n^k is $\Omega(c^n)$
- n^k is $\Theta(c^n)$

Answer

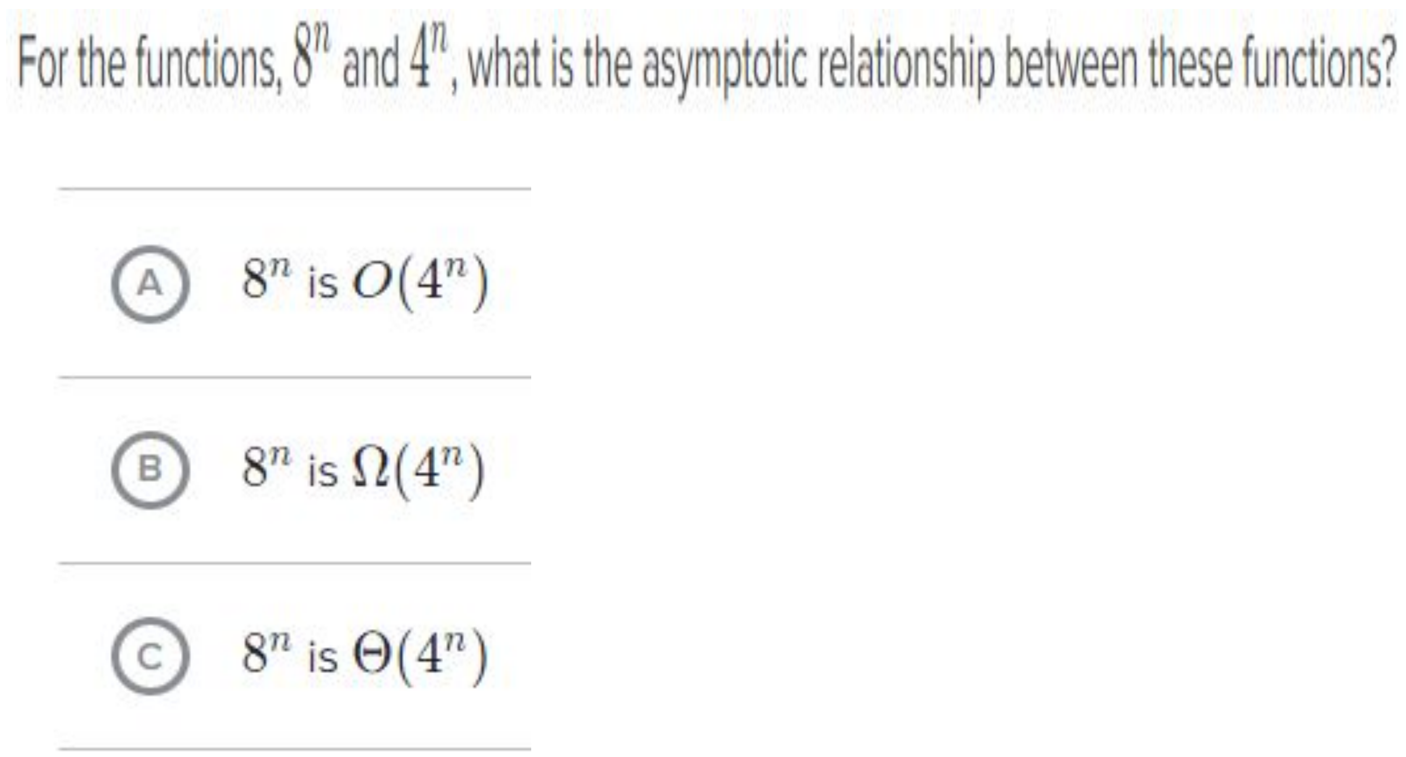
For the functions, n^k and $c^n c$, what is the asymptotic relationship between these functions? Assume that k >= 1 and c > 1 are constants.

- ☐ **n^k is $O(c^n)$**
- ☐ n^k is $\Omega(c^n)$
- ☐ n^k is $\Theta(c^n)$

n^k is a polynomial function, and c^n is a exponential function. We know that polynomials always grow more slowly than exponentials. We could prove that with a graph, but we have to make sure we look at it for big values of n, because the early behavior could be misleading. Here's a graph that compares the two functions (with k=2 and c=2), where we can clearly see the difference in growth:



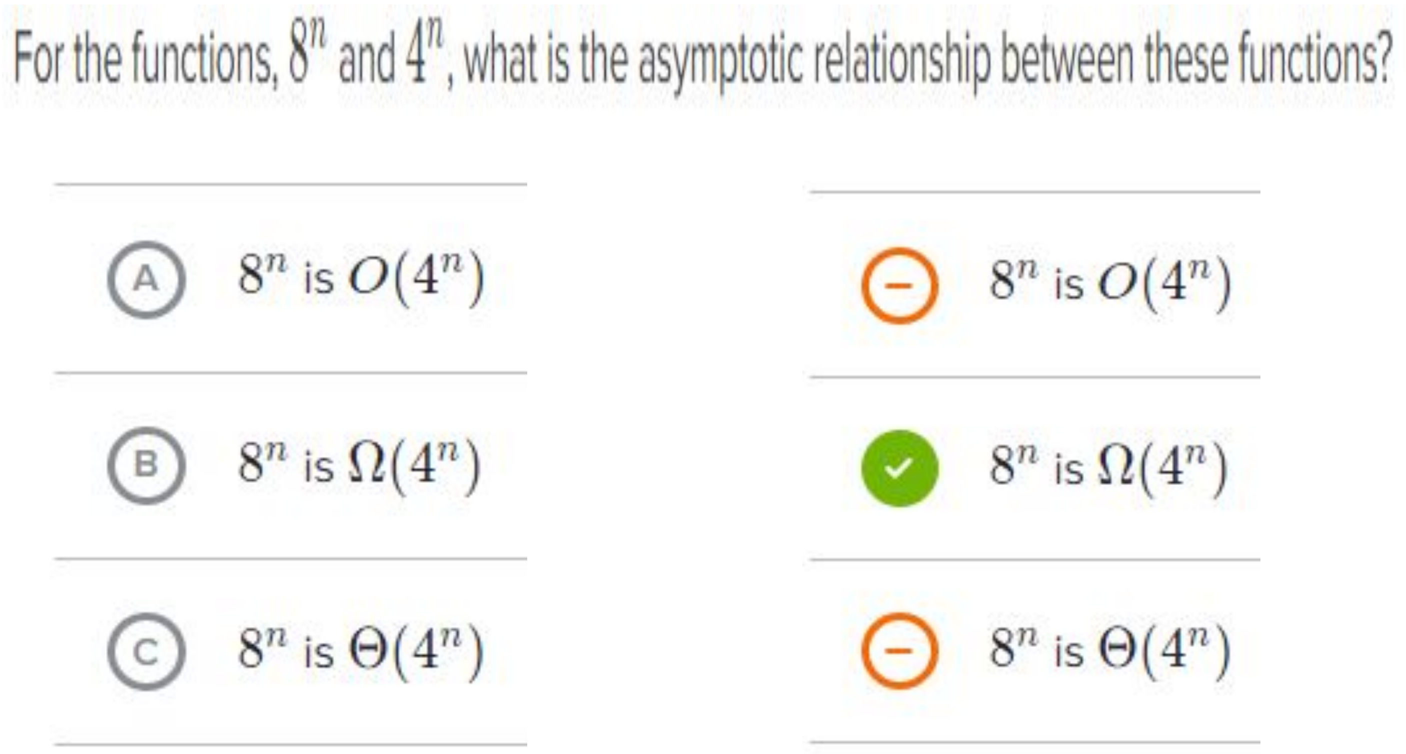
Question



For the functions, 8^n and 4^n, what is the asymptotic relationship between these functions?

- A 8^n is $O(4^n)$
- B 8^n is $\Omega(4^n)$
- C 8^n is $\Theta(4^n)$

Question

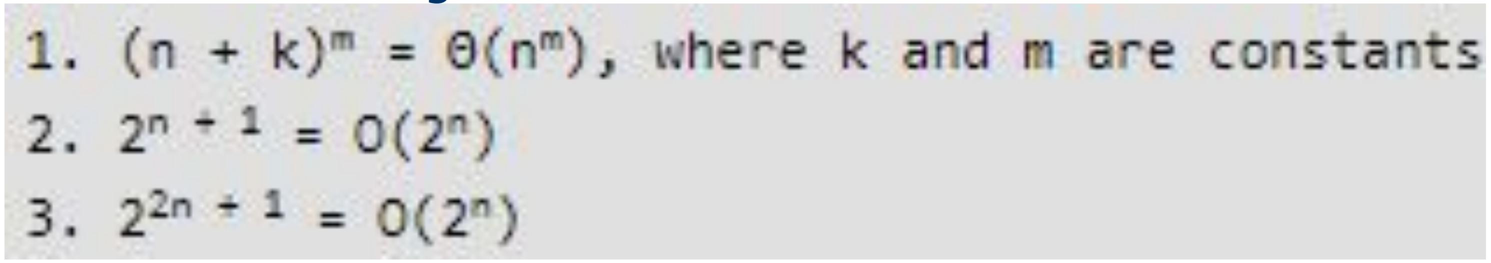


For the functions, 8^n and 4^n, what is the asymptotic relationship between these functions?

A	8^n is $O(4^n)$	⊖	8^n is $O(4^n)$
B	8^n is $\Omega(4^n)$	✓	8^n is $\Omega(4^n)$
C	8^n is $\Theta(4^n)$	⊖	8^n is $\Theta(4^n)$

Question

Consider the following three claims



1. $(n + k)^m = \Theta(n^m)$, where k and m are constants
2. $2^{n+1} = O(2^n)$
3. $2^{2n+1} = O(2^n)$

Which of these claims are correct ?

(A) 1 and 2 **(B)** 1 and 3
(C) 2 and 3 **(D)** 1, 2, and 3

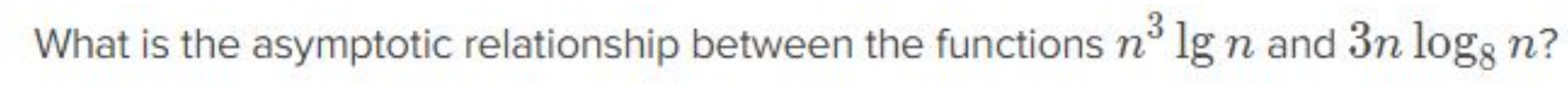
Answer:**(A)**

Explanation: $(n + k)^m$ and $\Theta(n^m)$ are asymptotically same as theta notation can always be written by taking the leading order term in a polynomial expression.

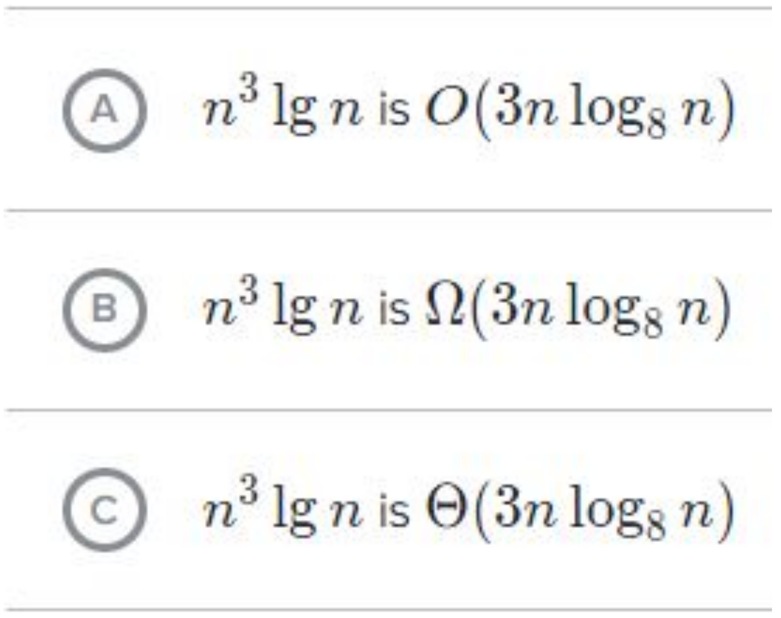
2^{n+1} and $O(2^n)$ are also asymptotically same as 2^{n+1} can be written as $2 * 2^n$ and constant multiplication/addition doesn't matter.

2^{2n+1} and $O(2^n)$ are not same as constant is in power.

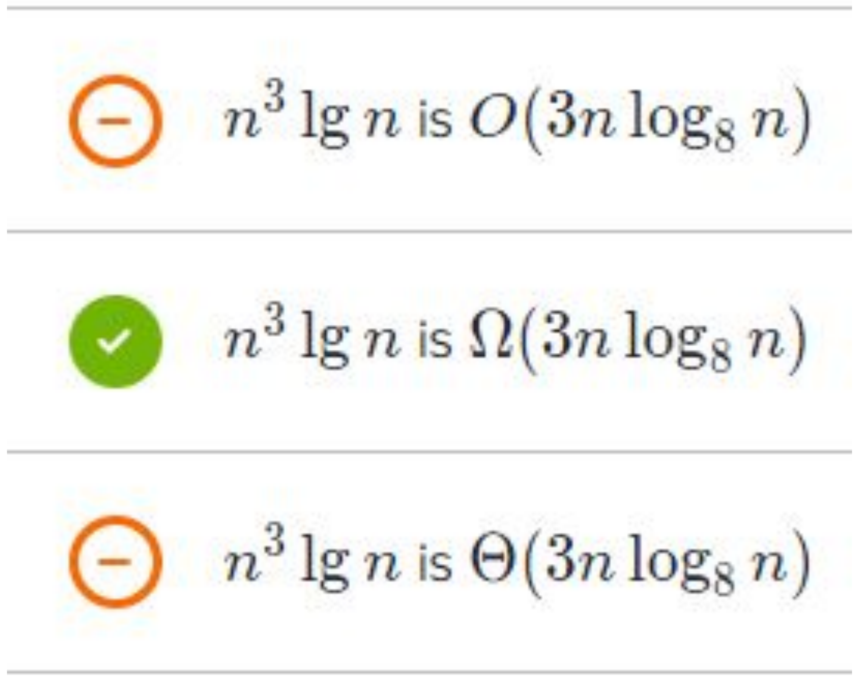
Question



What is the asymptotic relationship between the functions $n^3 \lg n$ and $3n \log_8 n$?

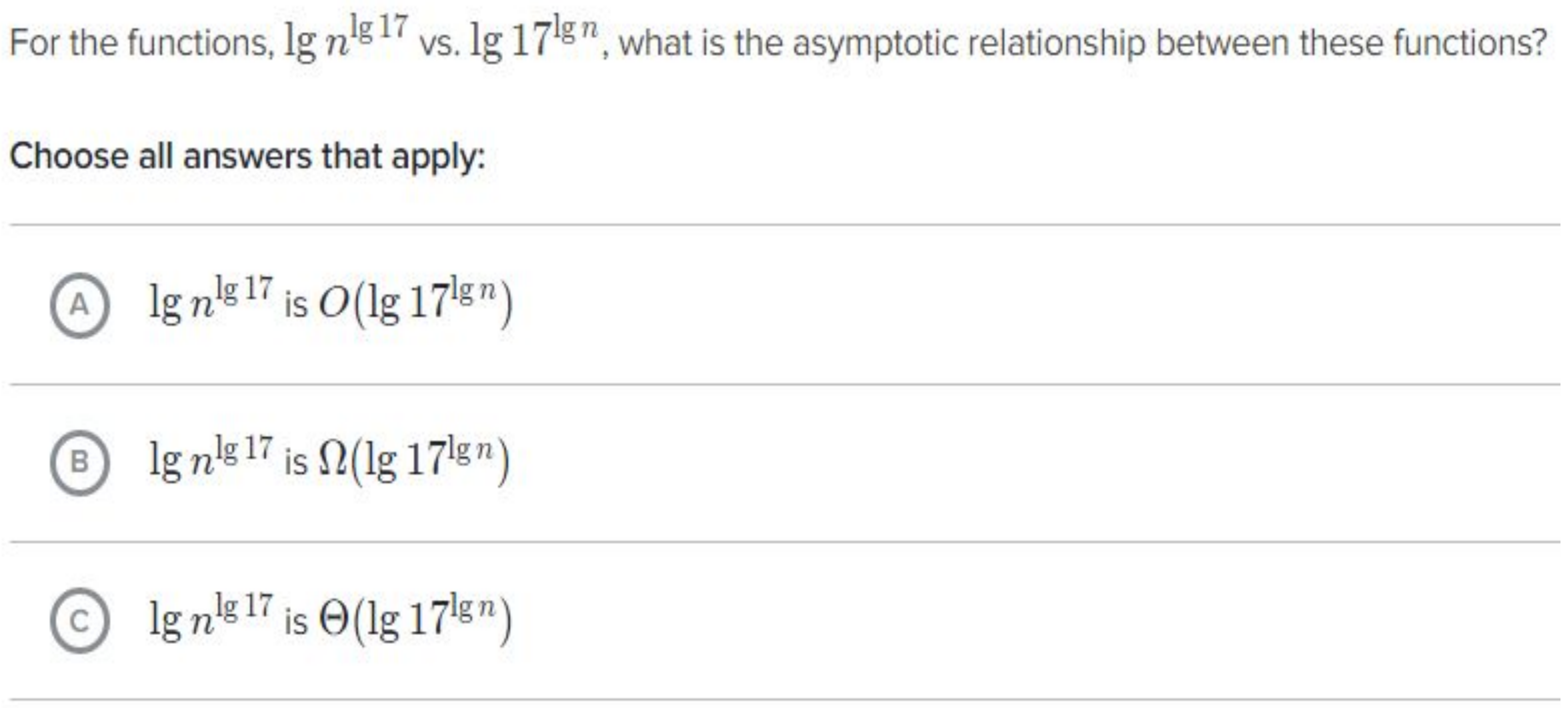


- (A) $n^3 \lg n$ is $O(3n \log_8 n)$
- (B) $n^3 \lg n$ is $\Omega(3n \log_8 n)$
- (C) $n^3 \lg n$ is $\Theta(3n \log_8 n)$



- ⊖ $n^3 \lg n$ is $O(3n \log_8 n)$
- [✓ $n^3 \lg n$ is $\Omega(3n \log_8 n)$]
- ⊖ $n^3 \lg n$ is $\Theta(3n \log_8 n)$

Question



For the functions, $\lg n^{\lg 17}$ vs. $\lg 17^{\lg n}$, what is the asymptotic relationship between these functions?

Choose all answers that apply:

(A) $\lg n^{\lg 17}$ is $O(\lg 17^{\lg n})$

(B) $\lg n^{\lg 17}$ is $\Omega(\lg 17^{\lg n})$

(C) $\lg n^{\lg 17}$ is $\Theta(\lg 17^{\lg n})$

Answer

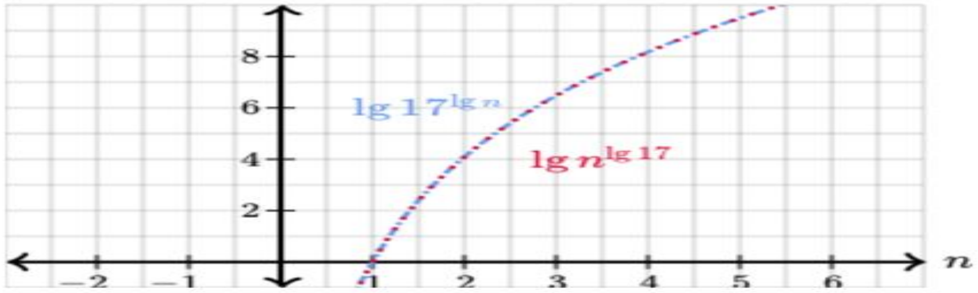
In conclusion, all of the statements are true.

✓ $\lg n^{\lg 17}$ is $O(\lg 17^{\lg n})$

✓ $\lg n^{\lg 17}$ is $\Omega(\lg 17^{\lg n})$

✓ $\lg n^{\lg 17}$ is $\Theta(\lg 17^{\lg n})$

Both $\lg n^{\lg 17}$ vs. $\lg 17^{\lg n}$ are functions with logarithmic growth, and the same base. They differ in what they take the logarithm of: $n^{\lg 17}$ versus $17^{\lg n}$. Here's a graph of the two functions:



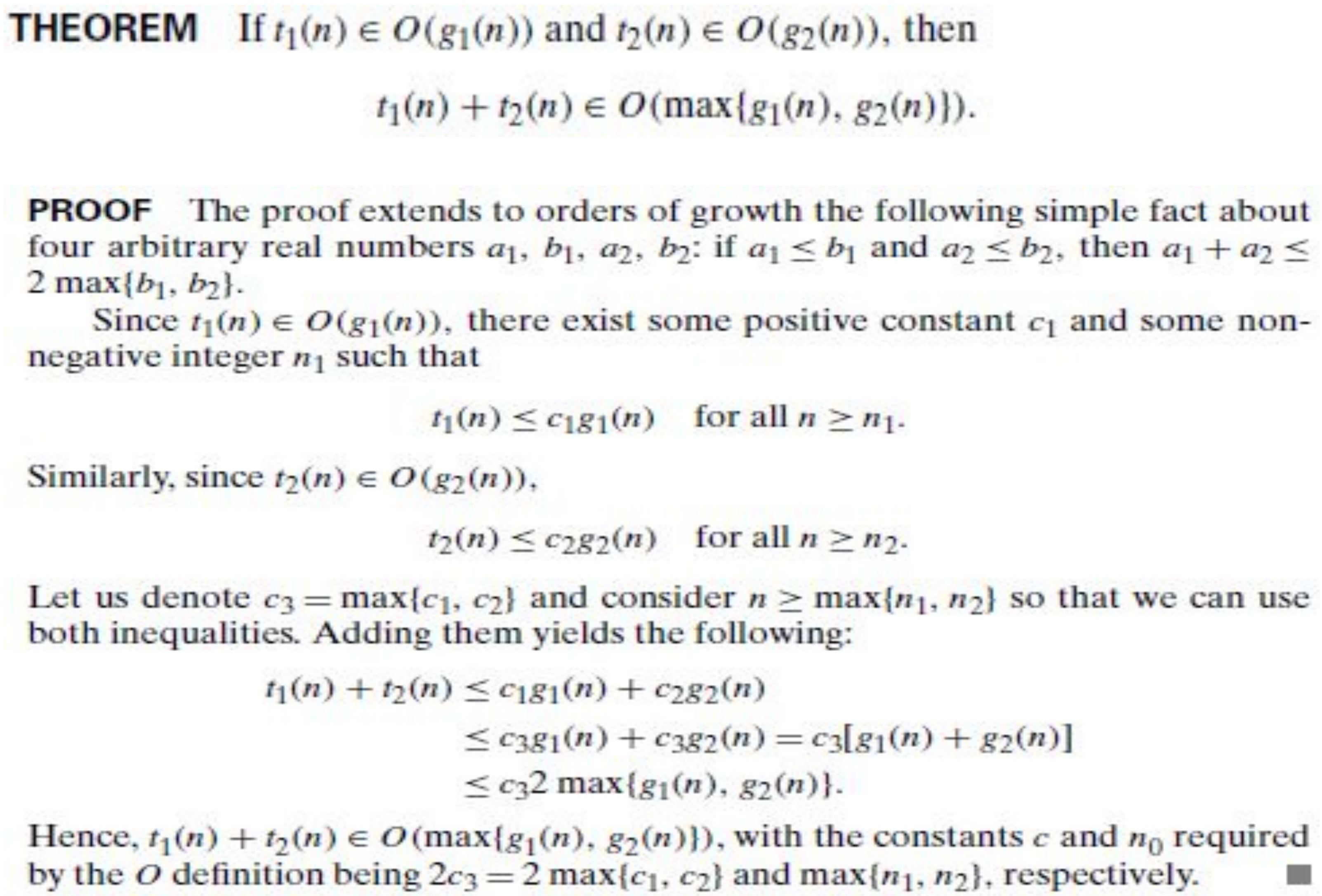
Notice something? It's the same graph! They're actually exactly equivalent functions, because of a particular property of logarithms:

$$\lg a^b = b \lg a$$

Let's re-write both of the original functions using that property:

original	becomes
$\lg\left(n^{(\lg 17)}\right)$	$\lg(17) \cdot \lg n$
$\lg\left(17^{\lg n}\right)$	$\lg(n) \cdot \lg(17)$

Useful Property Involving the Asymptotic Notations



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

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

PROOF The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2\max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some non-negative integer n_1 such that

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

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

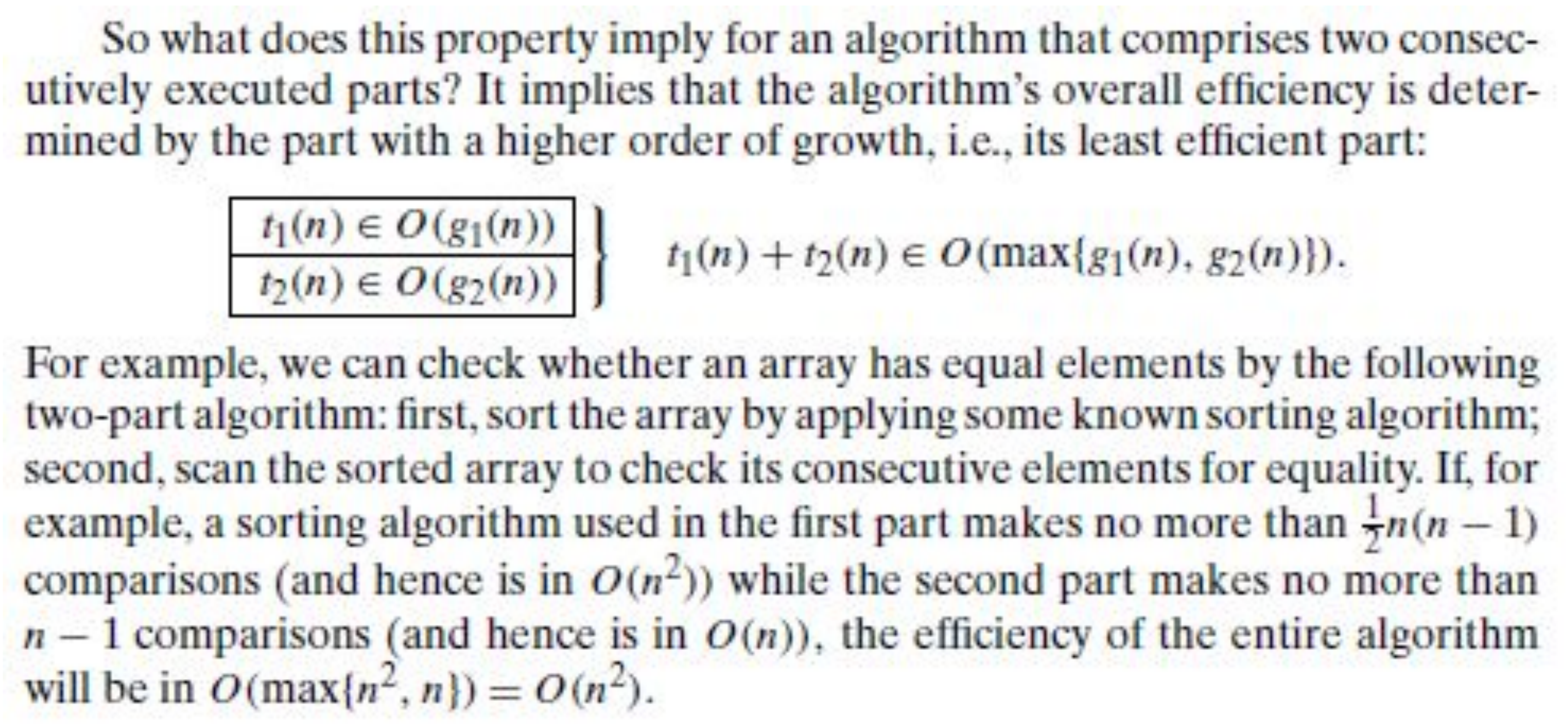
$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

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

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

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

Useful Property Involving the Asymptotic Notations (Contd…)



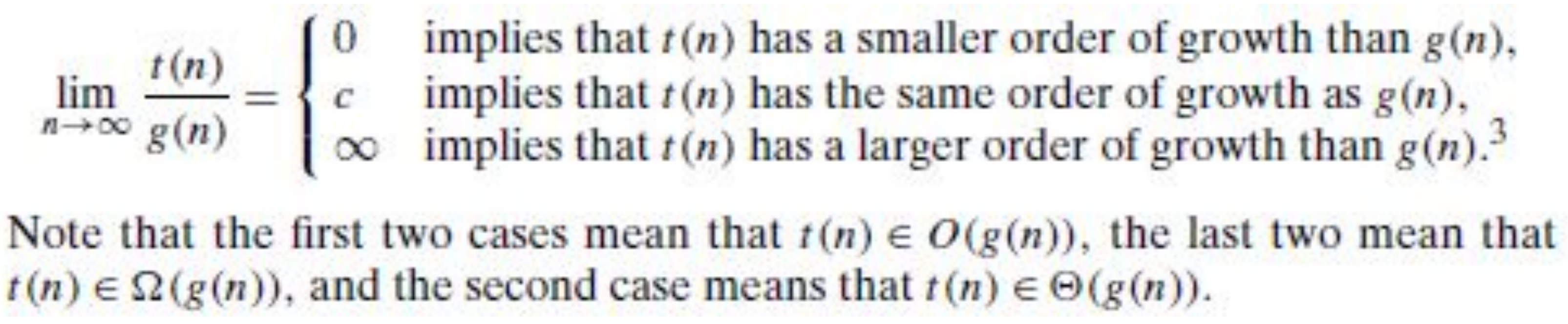
So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a higher order of growth, i.e., its least efficient part:

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

For example, we can check whether an array has equal elements by the following two-part algorithm: first, sort the array by applying some known sorting algorithm; second, scan the sorted array to check its consecutive elements for equality. If, for example, a sorting algorithm used in the first part makes no more than $\frac{1}{2}n(n-1)$ comparisons (and hence is in $O(n^2)$) while the second part makes no more than $n-1$ comparisons (and hence is in $O(n)$), the efficiency of the entire algorithm will be in $O(\max\{n^2, n\}) = O(n^2)$.

Using Limits for Comparing Orders of Growth

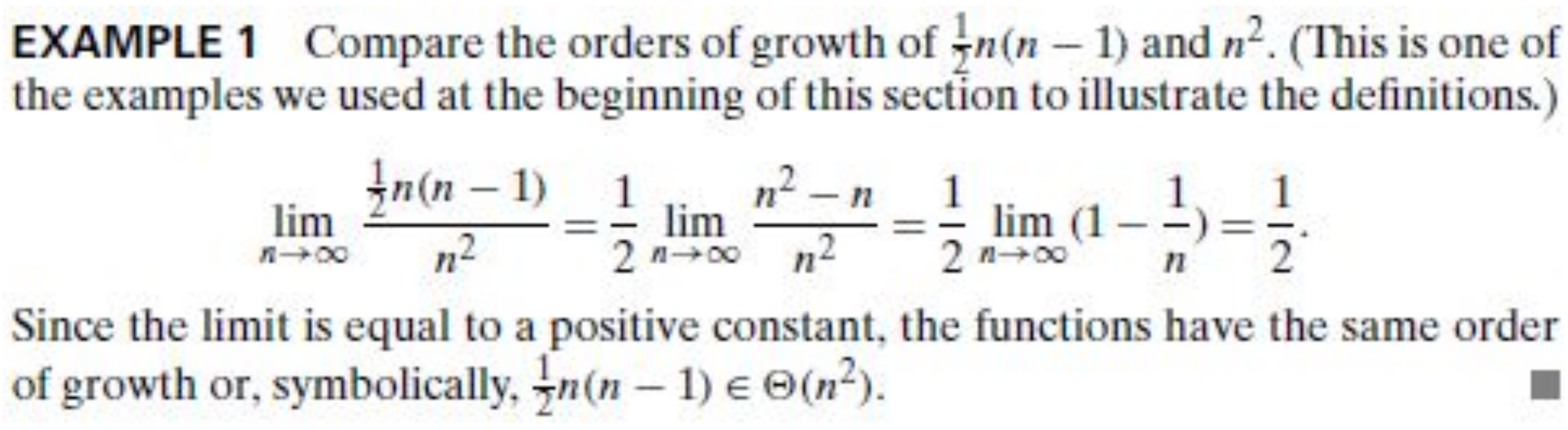
- Though the formal definitions of *O, Ω* and *Θ* are indispensable for proving their abstract properties, they are rarely used for comparing the orders of growth of two specific functions. A much more convenient method for doing so is based on computing the limit of the ratio of two functions in question. Three principal cases may arise:



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

Note that the first two cases mean that $t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, and the second case means that $t(n) \in \Theta(g(n))$.

Example



EXAMPLE 1 Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2. (This is one of the examples we used at the beginning of this section to illustrate the definitions.)

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

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

Problem

For each of the following functions, indicate the class $\Theta(g(n))$ the function belongs to. (Use the simplest $g(n)$ possible in your answers.) Prove your assertions.

a. $(n^2+1)^{10}$ b. $\sqrt{10n^2+7n+3}$

a. Informally, $(n^2+1)^{10} \approx (n^2)^{10} = n^{20} \in \Theta(n^{20})$ Formally,

$$\lim_{n\to\infty} \frac{(n^2+1)^{10}}{n^{20}} = \lim_{n\to\infty} \frac{(n^2+1)^{10}}{(n^2)^{10}} = \lim_{n\to\infty} \left(\frac{n^2+1}{n^2}\right)^{10} == \lim_{n\to\infty} \left(1+\frac{1}{n^2}\right)^{10} = 1.$$

Hence $(n^2+1)^{10} \in \Theta(n^{20})$.

b. Informally, $\sqrt{10n^2+7n+3} \approx \sqrt{10n^2} = \sqrt{10}n \in \Theta(n)$. Formally,

$$\lim_{n\to\infty} \frac{\sqrt{10n^2+7n+3}}{n} = \lim_{n\to\infty} \sqrt{\frac{10n^2+7n+3}{n^2}} = \lim_{n\to\infty} \sqrt{10+\frac{7}{n}+\frac{3}{n^2}} = \sqrt{10}.$$

Hence $\sqrt{10n^2+7n+3} \in \Theta(n)$.

Unit 1:

Mathematical analysis of

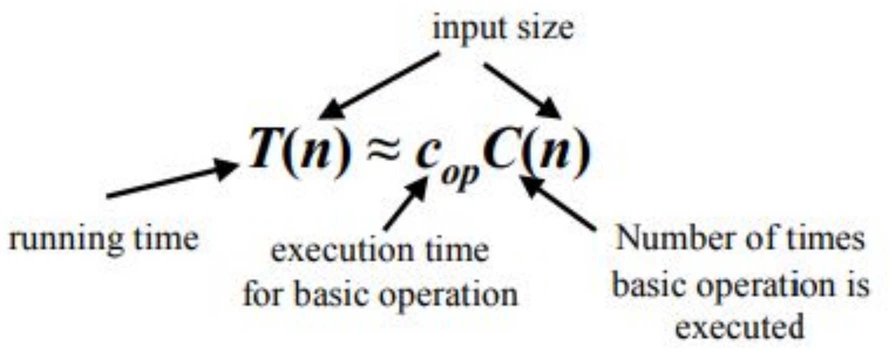
Non-recursive and Recursive Algorithms

Unit 1: Mathematical analysis of Non-recursive Algorithms

- General framework for analyzing time efficiency of Non-Recursive algorithms

Theoretical analysis of time efficiency

- Time efficiency is analyzed by determining the number of repetitions of the **basic operation** as a function of **input size.**
- **Basic operation:** the operation that contributes most towards the running time of the algorithm.

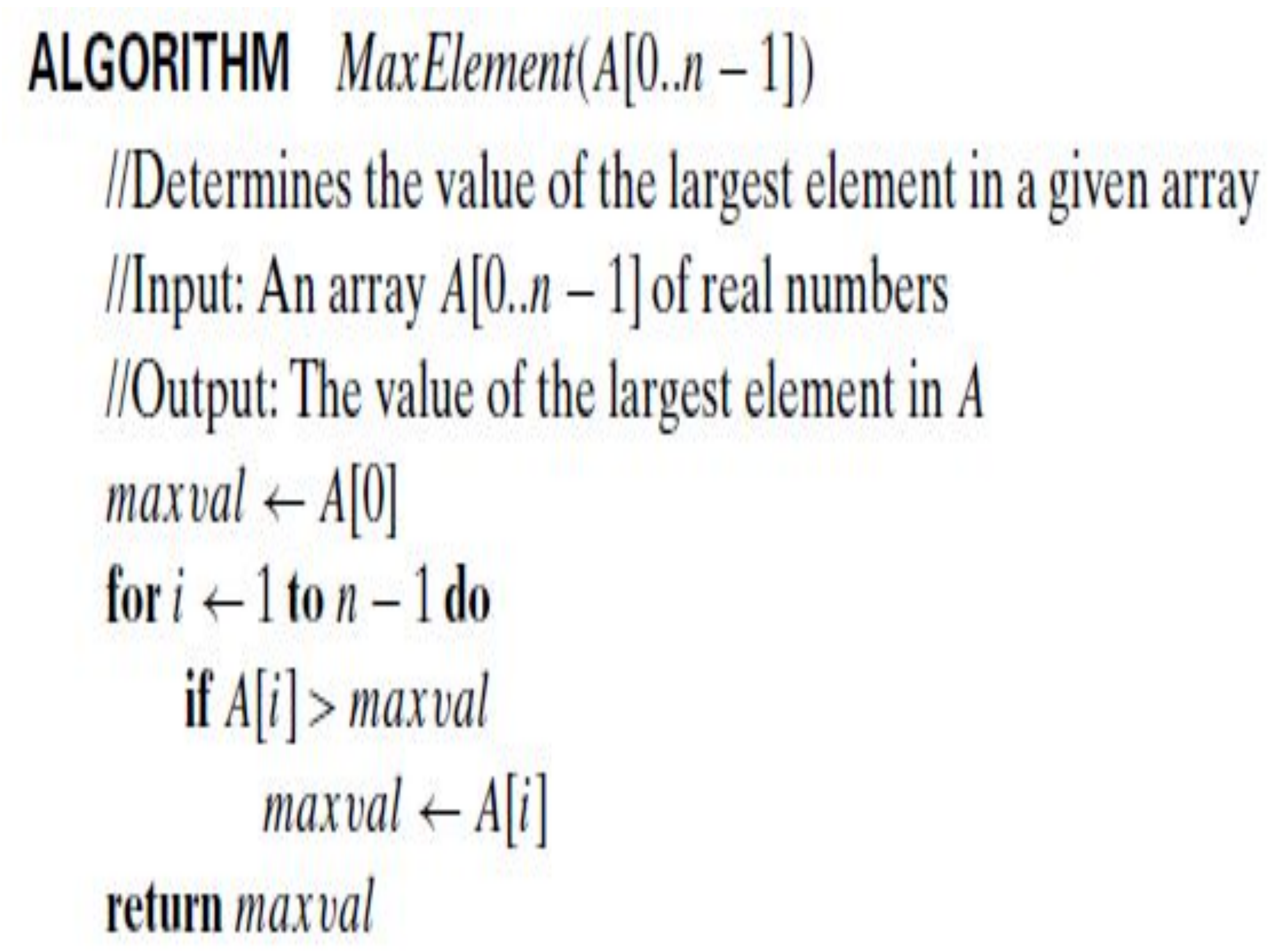

input size
$T(n) \approx c_{op}C(n)$
running time
execution time for basic operation
Number of times basic operation is executed

- An algorithm to solve a particular task employs some set of basic operations. When we estimate the amount of work done by an algorithm we usually do not consider all the steps such as e.g. initializing certain variables. Generally, the total number of steps is roughly proportional to the number of the basic operations. Thus, we are concerned mainly with the basic operations - **how many times the basic operations have to be performed depending on the size of input**.

Input size and Basic Operation examples

Problem	Input Size	Basic Operation
Search for key in list of n items	Number of items in list n	Comparison of Key element with Array element
Sort an array of numbers	The number of elements in the array	Comparison of two array entries
Multiply two matrices of floating point numbers	Dimensions of matrices	Floating point multiplication
Compute a^n	n	Floating point multiplication
Graph problem	Number of vertices and edges	Visiting a vertex or traversing an edge

Example: Maximum element in an Array



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

Example: Maximum element in an Array

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

- Basic Operation: Comparison operation A[i] > maxval
- Let us denote *T(n)* the number of times this comparison is executed and try to find a formula expressing it as a function of size *n*. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable *i* within the bounds 1 and *n* − 1, inclusive. Therefore, we get the following sum for

$$T(n) = \sum_{i=1}^{n-1} 1.$$

This is an easy sum to compute because it is nothing other than 1 repeated *n* − 1 times. Thus,

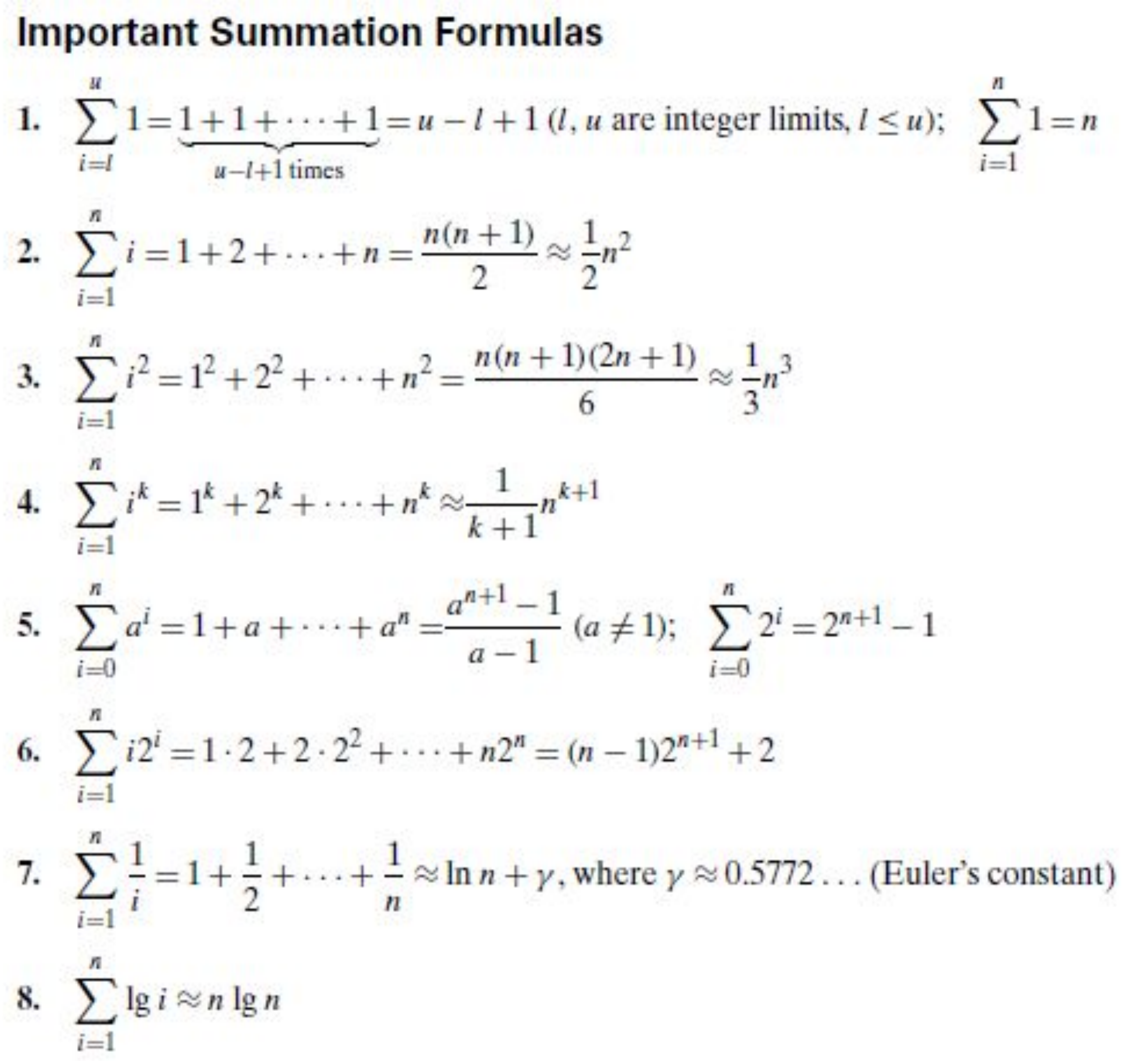
$$T(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Mathematical analysis of nonrecursive algorithms

Steps in mathematical analysis of nonrecursive algorithms:

- Decide on parameter n indicating input size
- Identify algorithm’s basic operation
- Determine worst, average, and best case for input of size n
- Set up summation for C(n) reflecting algorithm’s loop structure
- Simplify summation using standard formulas

Useful Formulas for the Analysis of Algorithms



Important Summation Formulas

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

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

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

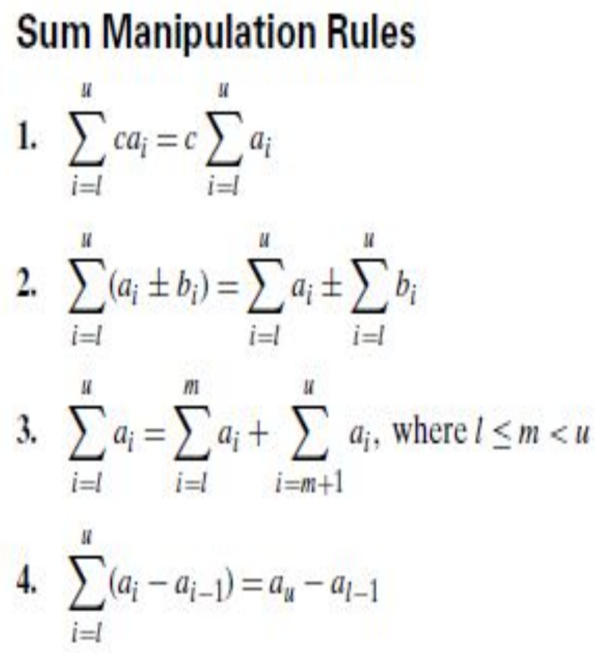
4. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1}n^{k+1}$

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

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

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

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



Sum Manipulation Rules

1. $\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i$

2. $\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$

3. $\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i$, where $l \le m < u$

4. $\sum_{i=l}^{u} (a_i - a_{i-1}) = a_u - a_{l-1}$

Commonly used sum manipulation rules and Summation formulas

- Two frequently used basic rules of sum manipulation

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u}(a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

- Summation Formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits,} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

Question

Consider the Algorithm

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

a. What does this algorithm compute?

b. What is its basic operation?

c. How many times is the basic operation executed?

d. What is the efficiency class of this algorithm?

Question

Consider the Algorithm

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

a. What does this algorithm compute?

b. What is its basic operation?

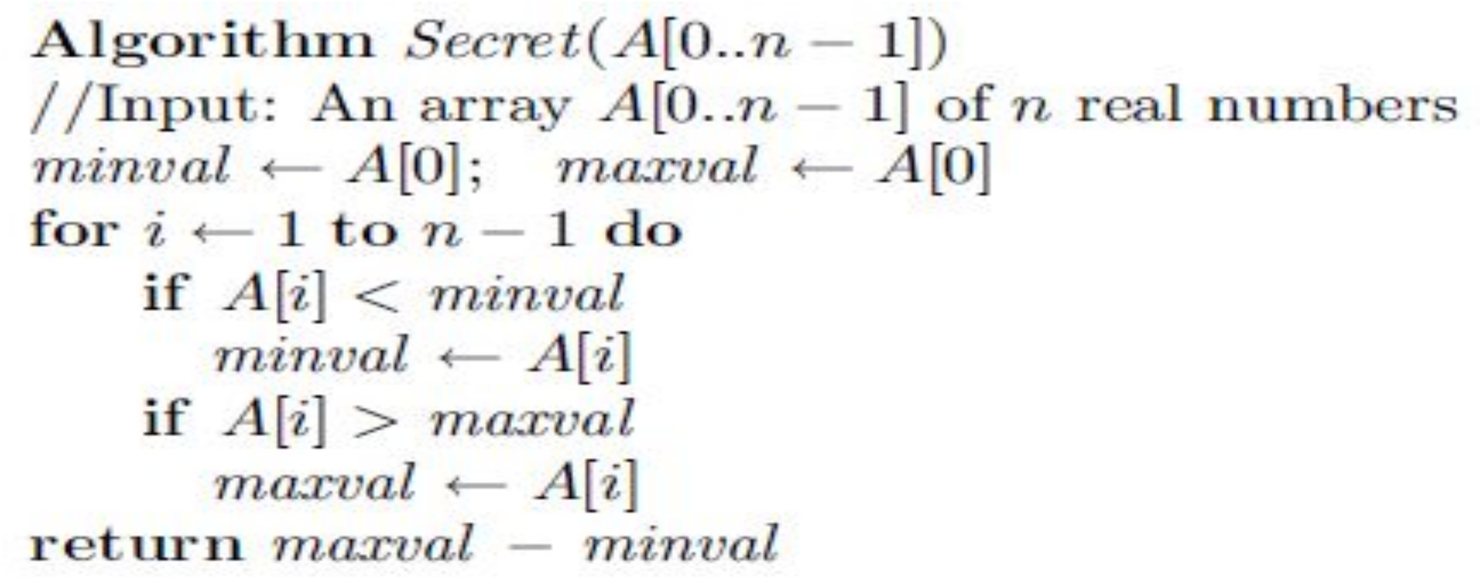
c. How many times is the basic operation executed?

d. What is the efficiency class of this algorithm?

Answer:

a. Computes the sum of the first n numbers
b. It is the summation
c. Number of executions = n
d. The basic operation is always(worst, average, best case) executed n times, so it's Θ (n).

Example: Finding the Maximum and Minimum Element in an array



```
Algorithm $Secret(A[0..n-1])$
//Input: An array $A[0..n-1]$ of $n$ real numbers
$minval \leftarrow A[0]$;   $maxval \leftarrow A[0]$
for $i \leftarrow 1$ to $n-1$ do
    if $A[i] < minval$
        $minval \leftarrow A[i]$
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval - minval$
```

a. What does this algorithm compute?

b. What is its basic operation?

c. How many times is the basic operation executed?

d. What is the efficiency class of this algorithm?

Answer

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

a. What does this algorithm compute?

b. What is its basic operation?

c. How many times is the basic operation executed?

d. What is the efficiency class of this algorithm?

a. Finding maximum and minimum

b. Basic operations: A[i] > minval and A[i] > maxval

c. d. $T(n)=\sum_{i=1}^{i=n-1} 2 = 2\sum_{i=1}^{i=n-1} 1 = 2(n-1) = 2n - 2 = \Theta(n)$

Question

- In the following code find out how many times the **sum++** will be executed

```
sum = 0;
for( i = 1; i <= n; i++) {
for( j = 1; j <= n; j++)
    { sum++; }
}
```

Answer

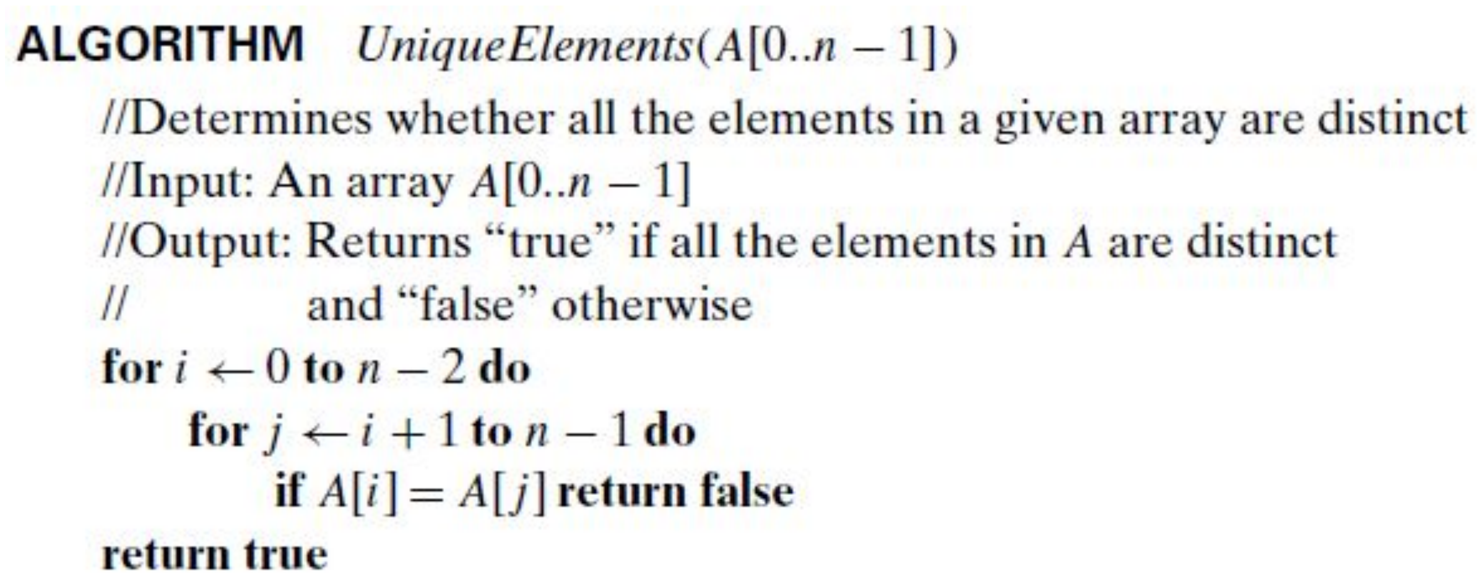
- In the following code find out how many times the **sum++** will be executed

```
sum = 0;
for( i = 1; i <= n; i++) {
for( j = 1; j <= n; j++)
      { sum++; }
}
```

$$T(n)=\sum_{i=1}^{i=n}\sum_{j=1}^{j=n} 1 = \Theta(n^2)$$

Example: Checking all elements in an array are distinct

- ***Element uniqueness problem***: check whether all the elements in a given array of *n* elements are distinct.



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

Example: Checking all elements in an array are distinct

- ***Element uniqueness problem***: check whether all the elements in a given array of *n* elements are distinct.

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

$$T(n) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \sum_{i=0}^{n-2}(n-1-i)$$

$$= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$$

We also could have computed the sum $\sum_{i=0}^{n-2}(n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1)+(n-2)+\cdots+1 = \frac{(n-1)n}{2},$$

Homework Problem

- In the following code find out how many times the **sum++** will be executed

```
sum = 0;
for( i = 1; i <= n; i++)
for( j = i; j <= n; j++)
    sum++;
```

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = \sum_{i=1}^{n} (n-i+1) = \sum_{i=1}^{n} (n+1) - \sum_{i=1}^{n} i =$$

$$n(n+1) - \frac{n(n+1)}{2} = \frac{n(n+1)}{2} \approx n^2$$

Homework Problem

- In the following code find out how many times the **sum++** will be executed

```
sum = 0;
for( i = 1; i <= n; i++)
for( j = 1; j <= 2n; j++)
    sum++;
```

Question

- In the following code find out how many times the **sum++** will be executed

```
for (i=1; i<=n*n; i++)
   for (j=0; j<i; j++)
      sum++;
```

Answer

```
for (i=1; i<=n*n; i++)
  for (j=0; j<i; j++)
    sum++;
```

Exact # of times `sum++` is executed:

$$\begin{aligned}\sum_{i=1}^{n^2} i &= \frac{n^2(n^2+1)}{2} \\ &= \frac{n^4+n^2}{2} \\ &\in \Theta(n^4)\end{aligned}$$

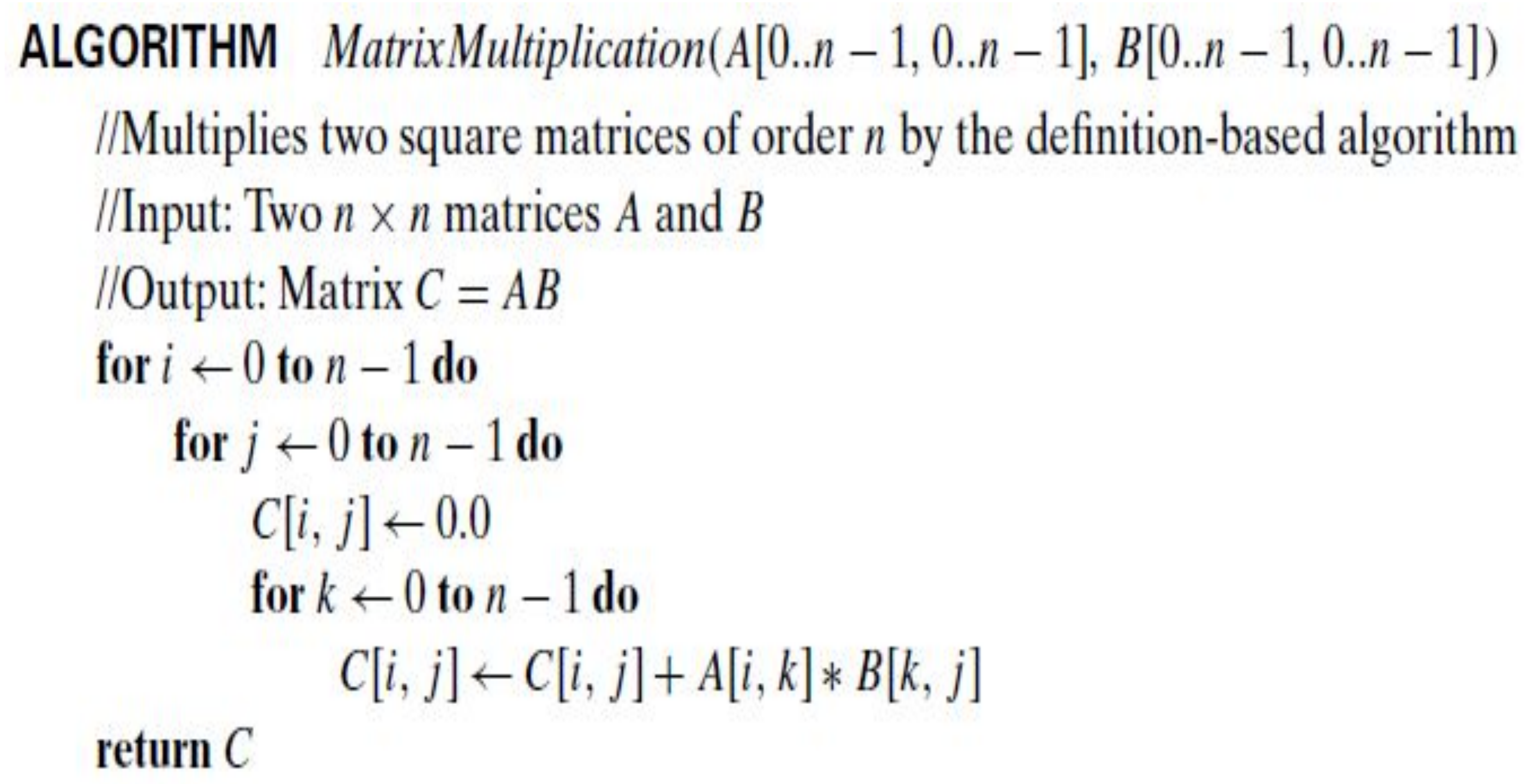
Homework Problem

In the following code find out how many times the **sum++** will be executed

```
sum = 0;
for( i = 0; i < n; i++)
for( j = 0; j < i*i; j++)
for( k = 0; k < j; k++)
   sum++;
```

Example: Two matrix Multiplication

- Given two $n \times n$ matrices *A* and *B,* find the time efficiency of the definition-based algorithm for computing their product $C = AB$



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

Example: Two matrix Multiplication

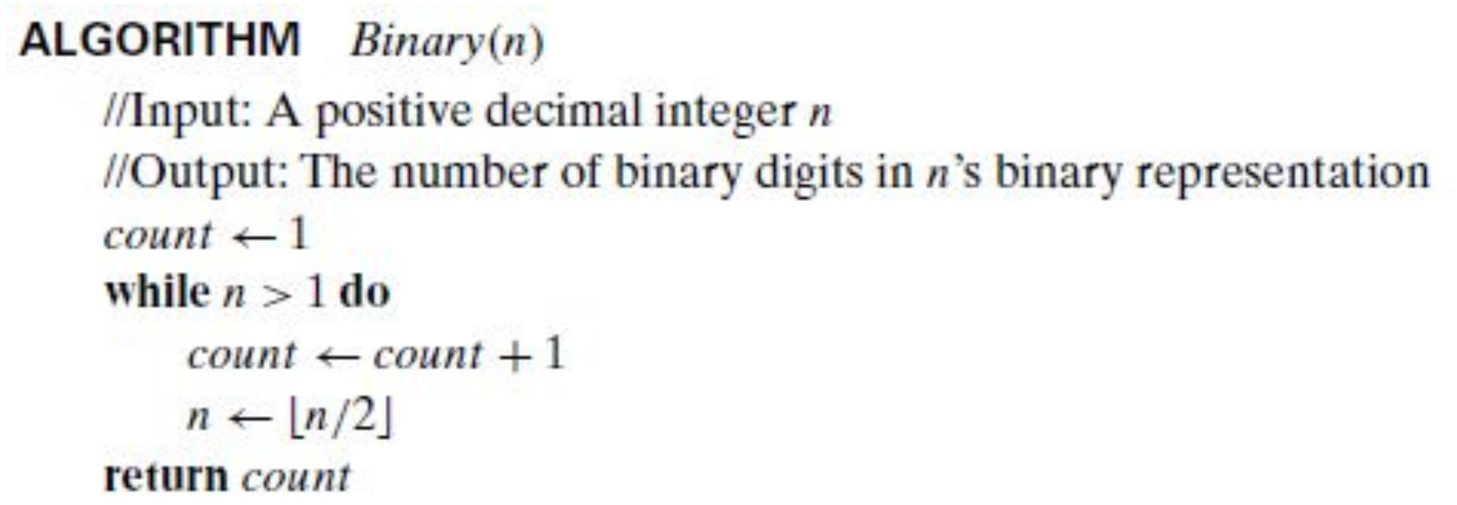
- Given two $n \times n$ matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$

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

$$T(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = \Theta(n^3)$$

Example: Counting binary digits in binary representation of a decimal number

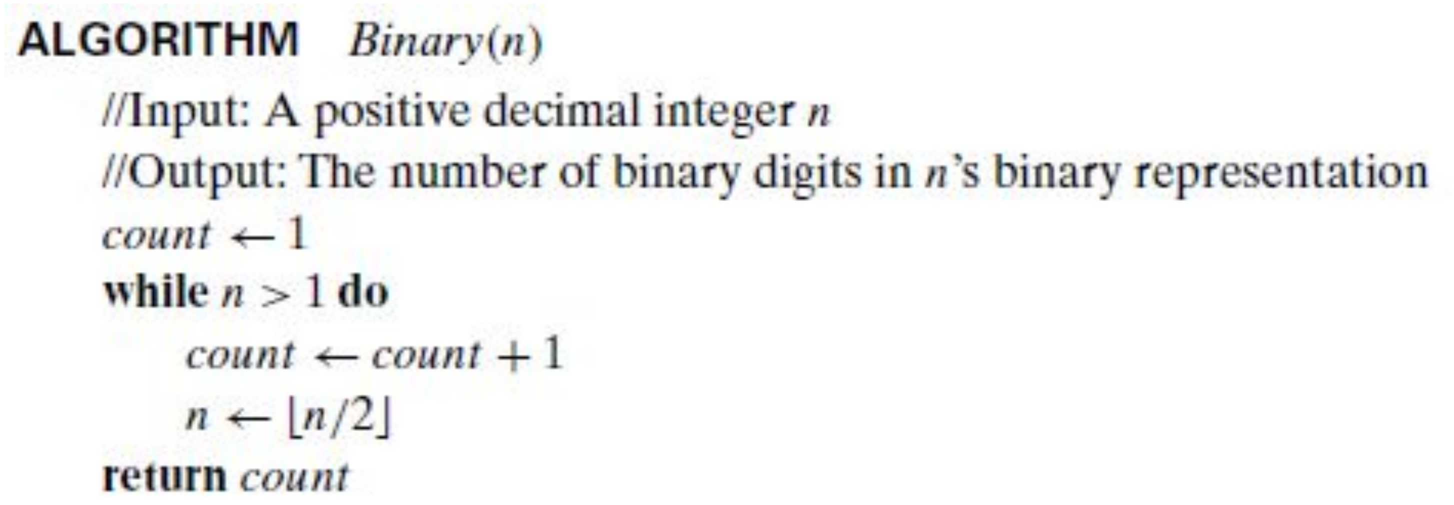
The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.



```
ALGORITHM  Binary(n)
    //Input: A positive decimal integer n
    //Output: The number of binary digits in n's binary representation
    count ← 1
    while n > 1 do
        count ← count + 1
        n ← ⌊n/2⌋
    return count
```

Example: Counting binary digits in binary representation of a decimal number

The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.



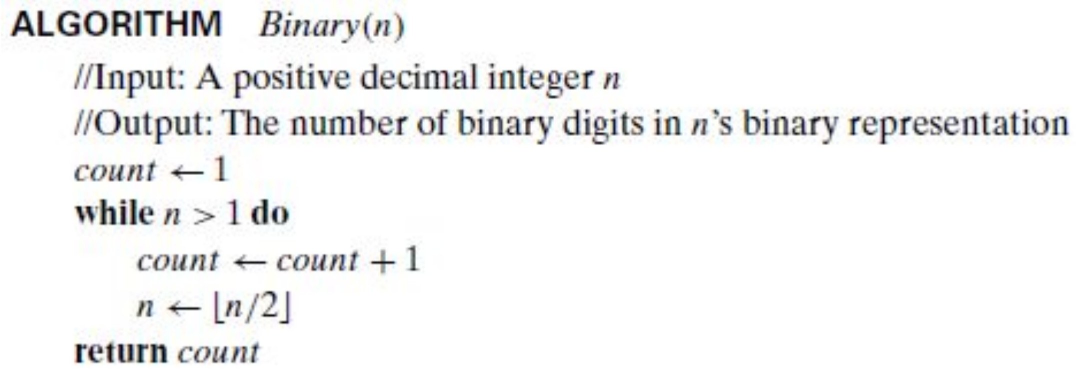
```
ALGORITHM  Binary(n)
    //Input: A positive decimal integer n
    //Output: The number of binary digits in n's binary representation
    count ← 1
    while n > 1 do
        count ← count + 1
        n ← ⌊n/2⌋
    return count
```

C(n) = n -> (n/2) -> (n/4) -> ....1 = $\lfloor \log_2 n \rfloor$

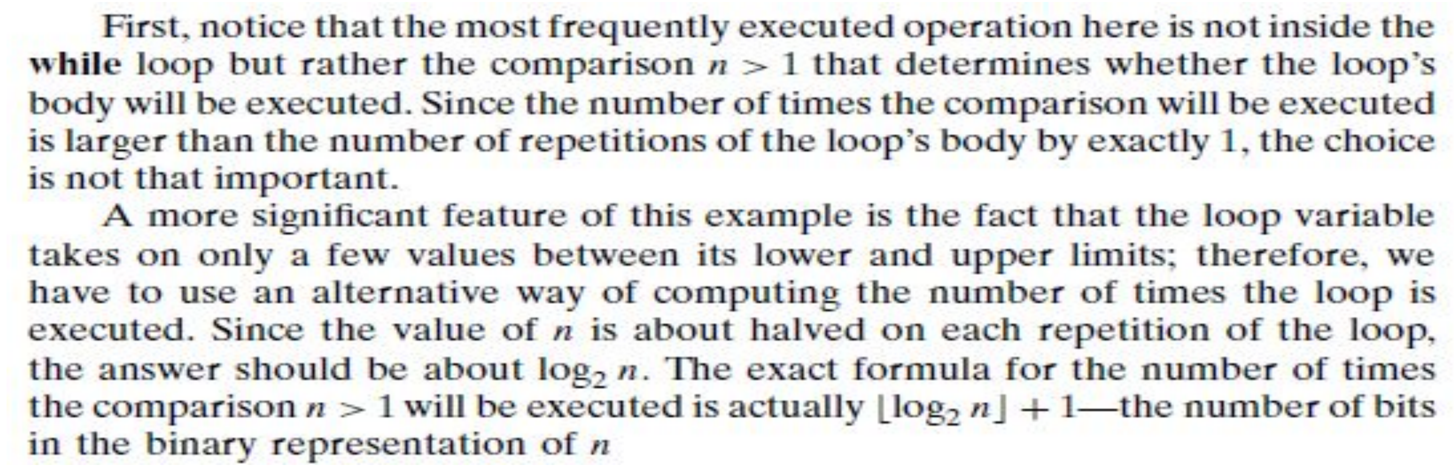
Number of times **n> 1** gets executed is $\lfloor \log_2 n \rfloor + 1$.

Example: Counting binary digits in binary representation of a decimal number

The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.



```
ALGORITHM  Binary(n)
    //Input: A positive decimal integer n
    //Output: The number of binary digits in n's binary representation
    count ← 1
    while n > 1 do
        count ← count + 1
        n ← ⌊n/2⌋
    return count
```



First, notice that the most frequently executed operation here is not inside the **while** loop but rather the comparison $n > 1$ that determines whether the loop's body will be executed. Since the number of times the comparison will be executed is larger than the number of repetitions of the loop's body by exactly 1, the choice is not that important.

A more significant feature of this example is the fact that the loop variable takes on only a few values between its lower and upper limits; therefore, we have to use an alternative way of computing the number of times the loop is executed. Since the value of n is about halved on each repetition of the loop, the answer should be about $\log_2 n$. The exact formula for the number of times the comparison $n > 1$ will be executed is actually $\lfloor \log_2 n \rfloor + 1$—the number of bits in the binary representation of n

Consecutive program fragments

- The total running time is the maximum of the running time of the individual fragments

```
sum = 0;
for( i = 0; i < n; i++)
    sum = sum + i;
sum = 0;
for( i = 0; i < n; i++)
     for( j = 0; j < 2n; j++)
         sum++;
```

The first loop runs in Θ (n) time,
the second Θ (n^2) time, the maximum is Θ (n^2)

Unit 1: Mathematical analysis of Recursive Algorithms

- General framework for analyzing time efficiency of Recursive algorithms

Time efficiency of recursive algorithms

Steps in mathematical analysis of recursive algorithms:

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- Determine worst, average, and best case for input of size n
- Set up a **recurrence relation and initial condition(s)** for T(n)-the number of times the basic operation will be executed for an input of size n (alternatively count recursive calls).
- Solve the recurrence to obtain a closed form or estimate the order of magnitude of the solution

Example: Factorial

- Analysis of recursive algorithm to find factorial of a given number

Example: Factorial

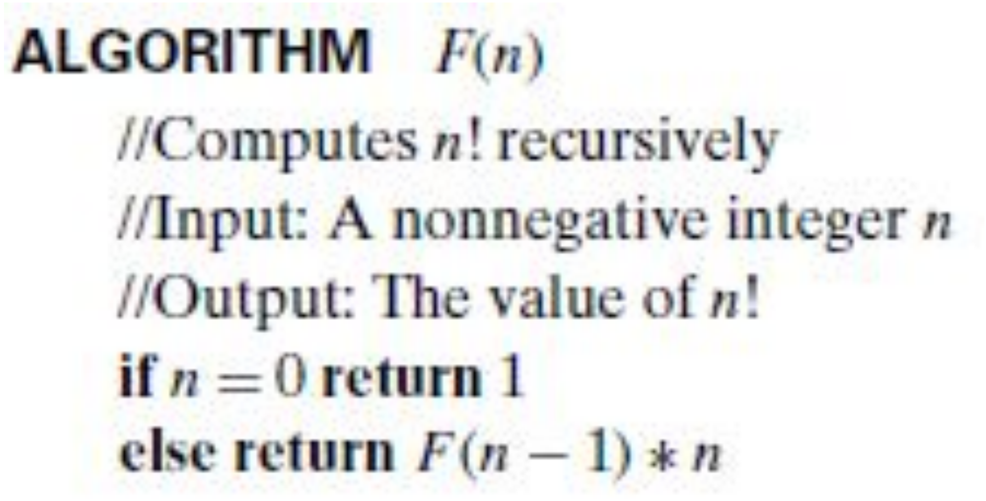
- Compute the factorial function $F(n) = n!$ for an arbitrary nonneg-ative integer n.Since

  $n! = 1 * 2 * \dots *(n - 1) *n = (n - 1)! \cdot n$ for $n \geq 1$

  and $0! = 1$ by definition, we can compute
- $F(n) = F(n - 1) \cdot n$ with the following recursive algorithm.

Example: Factorial

- Compute the factorial function ***F (n)*** = ***n***! for an arbitrary nonneg-ative integer ***n.***Since
  n! = 1 * 2 * ***… *(n*** − 1***) *n*** = ***(n*** − 1***)***! · ***n*** for ***n*** ≥ 1
  and 0! = 1 by definition, we can compute
- ***F(n)*** = ***F(n*** − 1***)*** · ***n*** with the following recursive algorithm.



```
ALGORITHM  $F(n)$
    //Computes $n!$ recursively
    //Input: A nonnegative integer $n$
    //Output: The value of $n!$
    if $n = 0$ return 1
    else return $F(n - 1) * n$
```

Example: Factorial

```
ALGORITHM  F(n)
    //Computes n! recursively
    //Input: A nonnegative integer n
    //Output: The value of n!
    if $n = 0$ return 1
    else return $F(n - 1) * n$
```

- The basic operation of the algorithm is multiplication, whose number of executions we denote *M(n).*

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute}\\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply}\\ F(n-1)\text{ by } n}} \quad \text{for } n > 0.$$

$$M(0) = 0.$$

the calls stop when $n = 0$ → $M(0)$ ← no multiplications when $n = 0$

Example: Factorial

```
ALGORITHM  F(n)
    //Computes n! recursively
    //Input: A nonnegative integer n
    //Output: The value of n!
    if $n = 0$ return 1
    else return $F(n-1) * n$
```

Recurrence relation and initial condition for the algorithm's number of multiplications *M(n)*:

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

$$M(0) = 0.$$

Example: Factorial

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

Solving the above recurrence relation using the *method of* ***backward substitutions***.

$$M(n) = M(n-1) + 1 \qquad \text{substitute } M(n-1) = M(n-2) + 1$$
$$= [M(n-2) + 1] + 1 = M(n-2) + 2 \qquad \text{substitute } M(n-2) = M(n-3) + 1$$
$$= [M(n-3) + 1] + 2 = M(n-3) + 3.$$
$$= M(n-i) + i$$
$$= M(n-n) + n$$

M(n)=n Hence the time complexity of recursive factorial algorithm is T(n) = ⊖(n)

Question

- Solve the following recurrence relations using Substitution method

a. $x(n) = x(n-1) + 5 \quad \text{for } n > 1, \quad x(1) = 0$

b. $x(n) = 3x(n-1) \quad \text{for } n > 1, \quad x(1) = 4$

d. $x(n) = x(n/2) + n \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 2^k)$

Answer

a. $x(n) = x(n-1) + 5 \quad \text{for } n > 1, \quad x(1) = 0$

$$\begin{aligned}
x(n) &= x(n-1) + 5 \\
&= [x(n-2) + 5] + 5 = x(n-2) + 5 \cdot 2 \\
&= [x(n-3) + 5] + 5 \cdot 2 = x(n-3) + 5 \cdot 3 \\
&= \ldots \\
&= x(n-i) + 5 \cdot i \\
&= \ldots \\
&= x(1) + 5 \cdot (n-1) = 5(n-1).
\end{aligned}$$

Note: The solution can also be obtained by using the formula for the n term of the arithmetical progression:

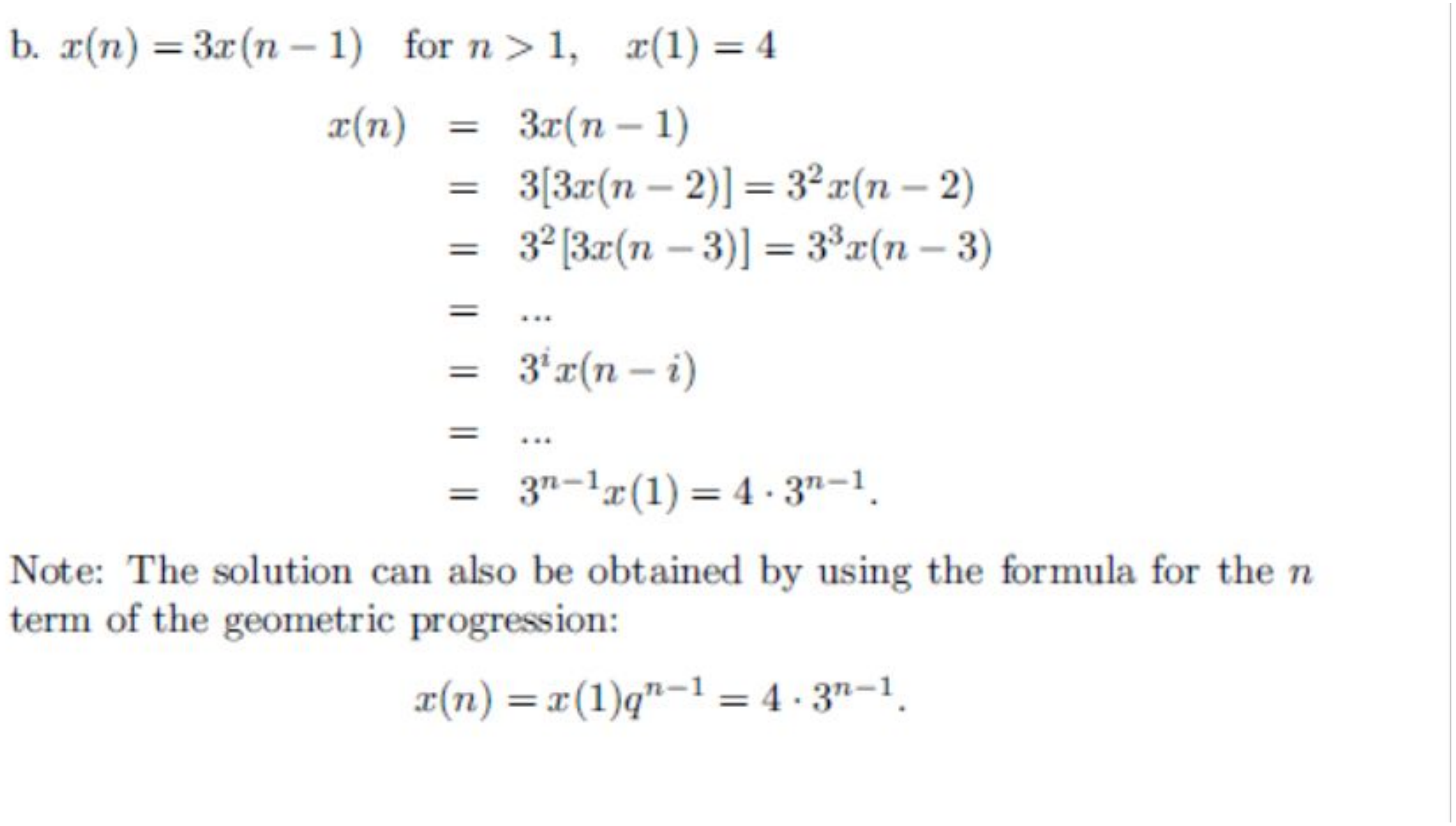
$$x(n) = x(1) + d(n-1) = 0 + 5(n-1) = 5(n-1).$$

Question

Solve the following recurrence relation

b. $x(n) = 3x(n-1)$ for $n > 1$, $x(1) = 4$

Answer



b. $x(n) = 3x(n-1) \quad \text{for } n > 1, \quad x(1) = 4$

$$\begin{aligned} x(n) &= 3x(n-1) \\ &= 3[3x(n-2)] = 3^2x(n-2) \\ &= 3^2[3x(n-3)] = 3^3x(n-3) \\ &= \dots \\ &= 3^ix(n-i) \\ &= \dots \\ &= 3^{n-1}x(1) = 4 \cdot 3^{n-1}. \end{aligned}$$

Note: The solution can also be obtained by using the formula for the n term of the geometric progression:

$$x(n) = x(1)q^{n-1} = 4 \cdot 3^{n-1}.$$

Question

Solve the following recurrence relations.

d. $x(n) = x(n/2) + n \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 2^k)$

Answer

d. $x(n) = x(n/2) + n \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 2^k)$

$$\begin{aligned}
x(2^k) &= x(2^{k-1}) + 2^k \\
&= [x(2^{k-2}) + 2^{k-1}] + 2^k = x(2^{k-2}) + 2^{k-1} + 2^k \\
&= [x(2^{k-3}) + 2^{k-2}] + 2^{k-1} + 2^k = x(2^{k-3}) + 2^{k-2} + 2^{k-1} + 2^k \\
&= \ldots \\
&= x(2^{k-i}) + 2^{k-i+1} + 2^{k-i+2} + \ldots + 2^k \\
&= \ldots \\
&= x(2^{k-k}) + 2^1 + 2^2 + \ldots + 2^k = 1 + 2^1 + 2^2 + \ldots + 2^k \\
&= 2^{k+1} - 1 = 2 \cdot 2^k - 1 = 2n - 1.
\end{aligned}$$

HomeWork Problem to Solve

Solve the following recurrence relations.

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

e. $x(n) = x(n/3) + 1$ for $n > 1$, $x(1) = 1$ (solve for $n = 3^k$)

Answer

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

$$\begin{aligned}
x(n) &= x(n-1) + n \\
&= [x(n-2) + (n-1)] + n = x(n-2) + (n-1) + n \\
&= [x(n-3) + (n-2)] + (n-1) + n = x(n-3) + (n-2) + (n-1) + n \\
&= \ldots \\
&= x(n-i) + (n-i+1) + (n-i+2) + \ldots + n \\
&= \ldots \\
&= x(0) + 1 + 2 + \ldots + n = \frac{n(n+1)}{2}.
\end{aligned}$$

e. $x(n) = x(n/3) + 1 \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 3^k)$

$$\begin{aligned}
x(3^k) &= x(3^{k-1}) + 1 \\
&= [x(3^{k-2}) + 1] + 1 = x(3^{k-2}) + 2 \\
&= [x(3^{k-3}) + 1] + 2 = x(3^{k-3}) + 3 \\
&= \ldots \\
&= x(3^{k-i}) + i \\
&= \ldots \\
&= x(3^{k-k}) + k = x(1) + k = 1 + \log_3 n.
\end{aligned}$$

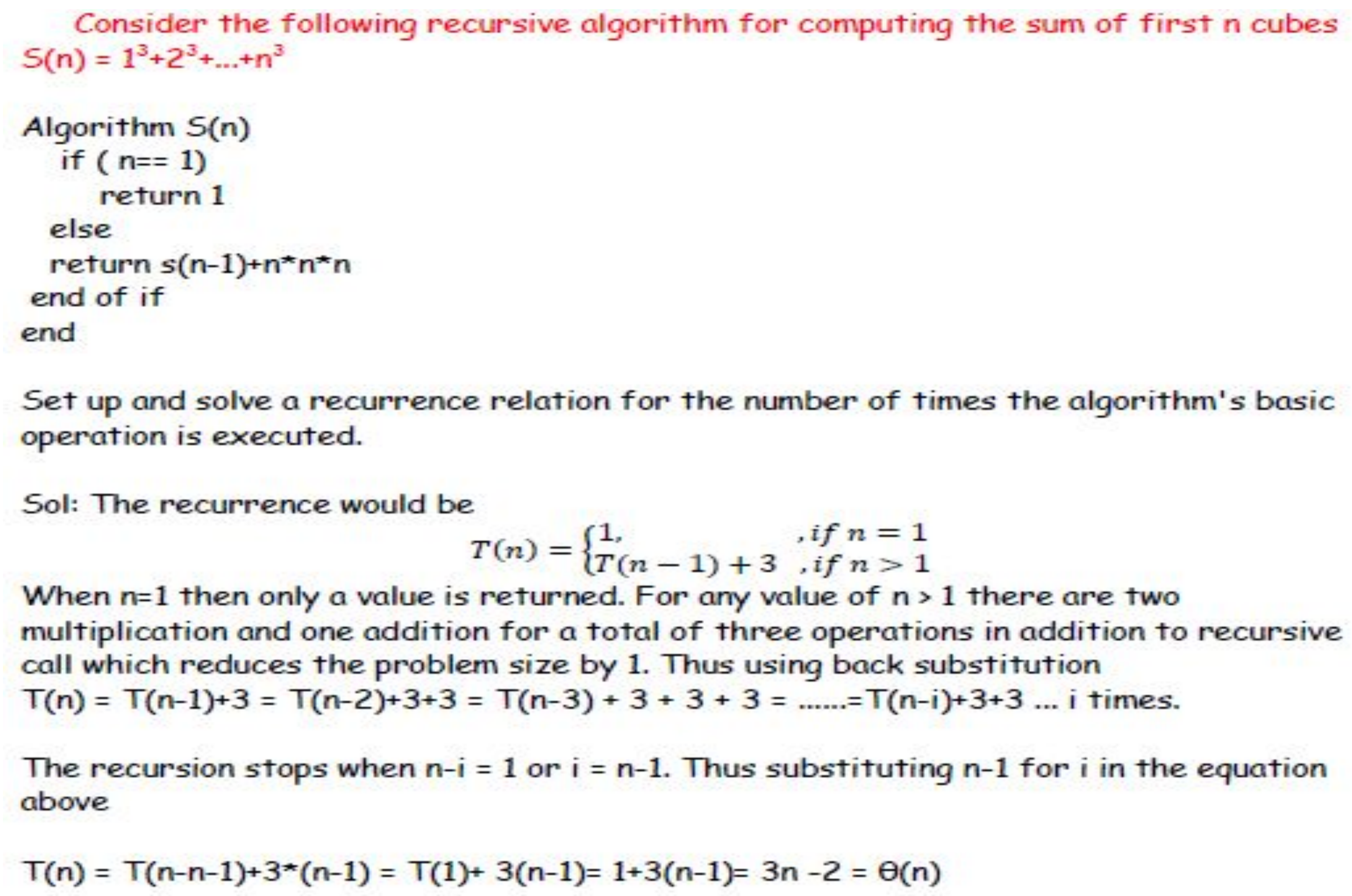
Homework Problem

Consider the following recursive algorithm for computing the sum of first n cubes $S(n) = 1^3+2^3+...+n^3$

```
Algorithm S(n)
   if ( n== 1)
      return 1
  else
  return s(n-1)+n*n*n
 end of if
end
```

Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.

Answer



Consider the following recursive algorithm for computing the sum of first n cubes
$S(n) = 1^3+2^3+...+n^3$

```
Algorithm S(n)
   if ( n== 1)
      return 1
  else
  return s(n-1)+n*n*n
 end of if
end
```

Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.

Sol: The recurrence would be

$$T(n) = \begin{cases} 1, & ,if\ n = 1 \\ T(n-1)+3 & ,if\ n > 1 \end{cases}$$

When n=1 then only a value is returned. For any value of n > 1 there are two multiplication and one addition for a total of three operations in addition to recursive call which reduces the problem size by 1. Thus using back substitution
$T(n) = T(n-1)+3 = T(n-2)+3+3 = T(n-3) + 3 + 3 + 3 = ...=T(n-i)+3+3$ ... i times.

The recursion stops when $n-i = 1$ or $i = n-1$. Thus substituting n-1 for i in the equation above

$T(n) = T(n-n-1)+3*(n-1) = T(1)+ 3(n-1)= 1+3(n-1)= 3n -2 = \Theta(n)$

Example 2:

- Towers of Hanoi

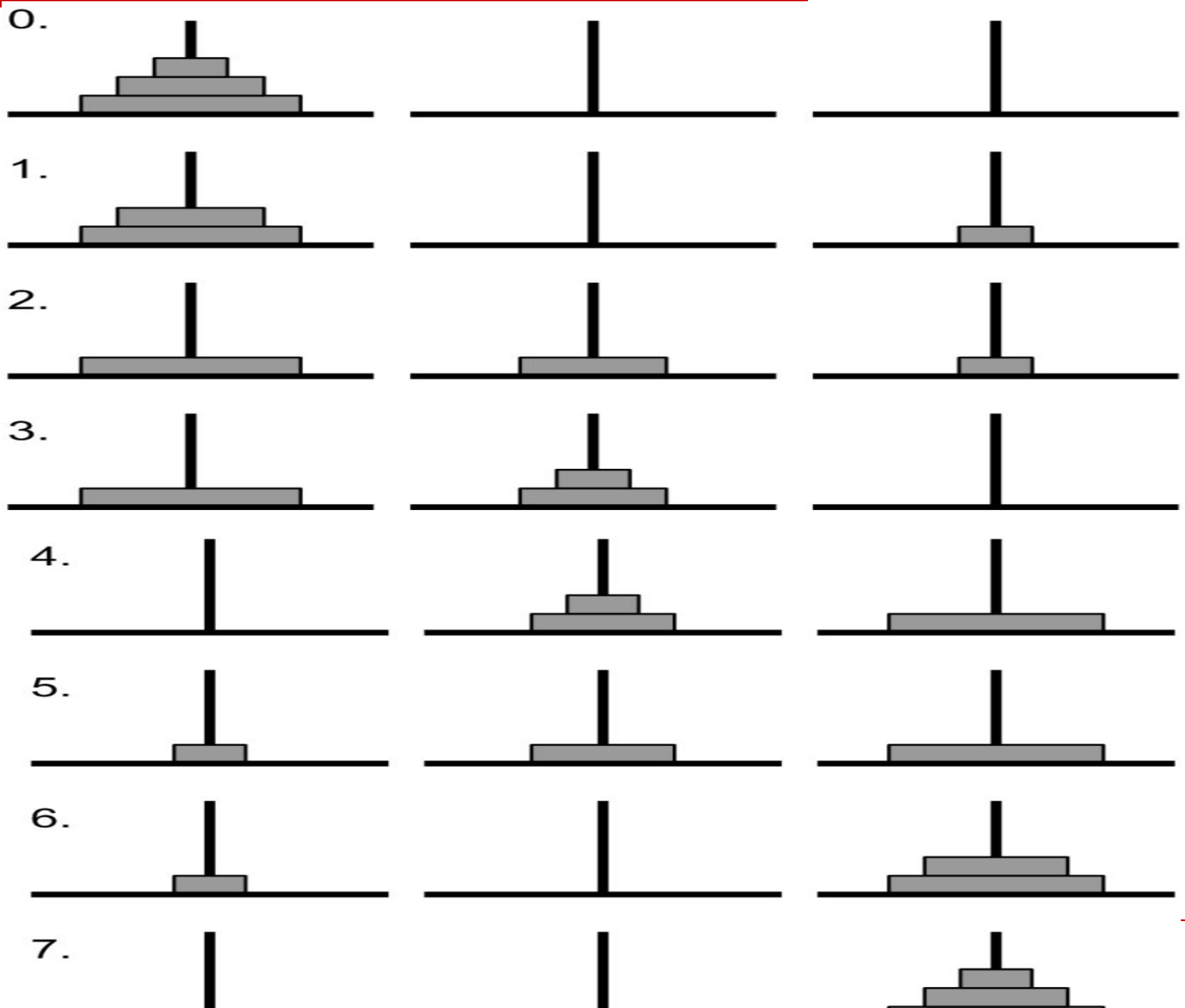
Example: Tower of Hanoi

- There are three pegs, Source(A), Auxiliary(B) and Destination(C). Peg A contains a set of disks stacked to resemble a tower, with the largest disk at the bottom and the smallest disk at the top. The objective is to transfer the entire tower of disks in peg A to peg C maintaining the same order of the disks.

Obeying the following rules:

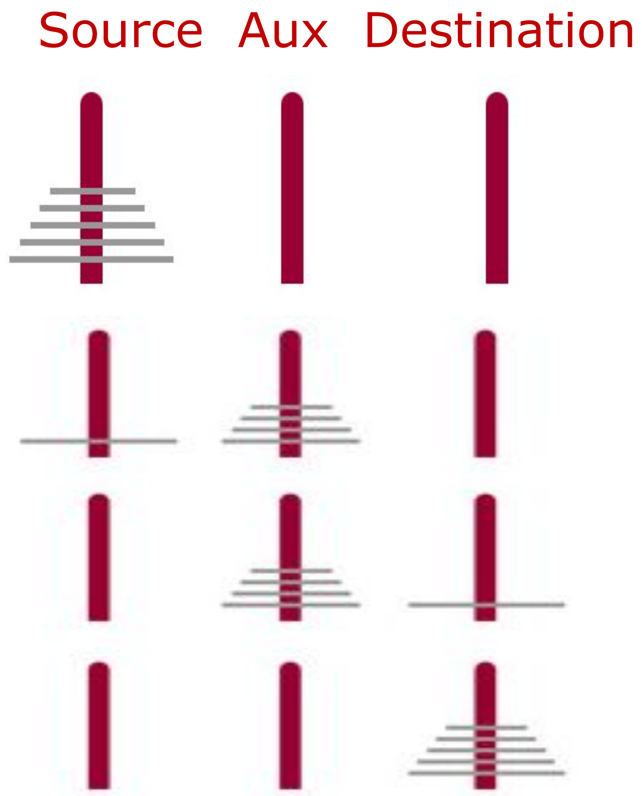
- Only one disk can be transfer at a time.
- Each move consists of taking the upper disk from one of the peg and placing it on the top of another peg i.e. a disk can only be moved if it is the uppermost disk of the peg.
- Never a larger disk is placed on a smaller disk during the transfer.

Tower of Hanoi: 3 Discs



Towers of Hanoi: Recursive

- Move **(n-1)** discs from the source post to the auxiliary post.
- Move the **last** disc to the destination post.
- Move **(n-1)** discs back from the auxiliary post to the destination post.



Towers of Hanoi: Recursive

```
Algorithm TowerHanoi(n, source, dest, aux)
IF n == 1, THEN
    move disk from source to dest
ELSE
     TowerHanoi(n - 1, source, aux, dest)
     move disk from source to dest
     TowerHanoi(n - 1, aux, dest, source)
END IF
END
```

Towers of Hanoi: Recursive

```
Algorithm TowerHanoi(n, source, dest, aux)
IF n == 1, THEN
    move disk from source to dest
ELSE
TowerHanoi(n - 1, source, aux, dest)
move disk from source to dest
TowerHanoi(n - 1, aux, dest, source)
END IF
END
```

The number of disks n is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. Clearly, the number of moves $M(n)$ depends on n only, and we get the following recurrence equation for it:

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

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

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

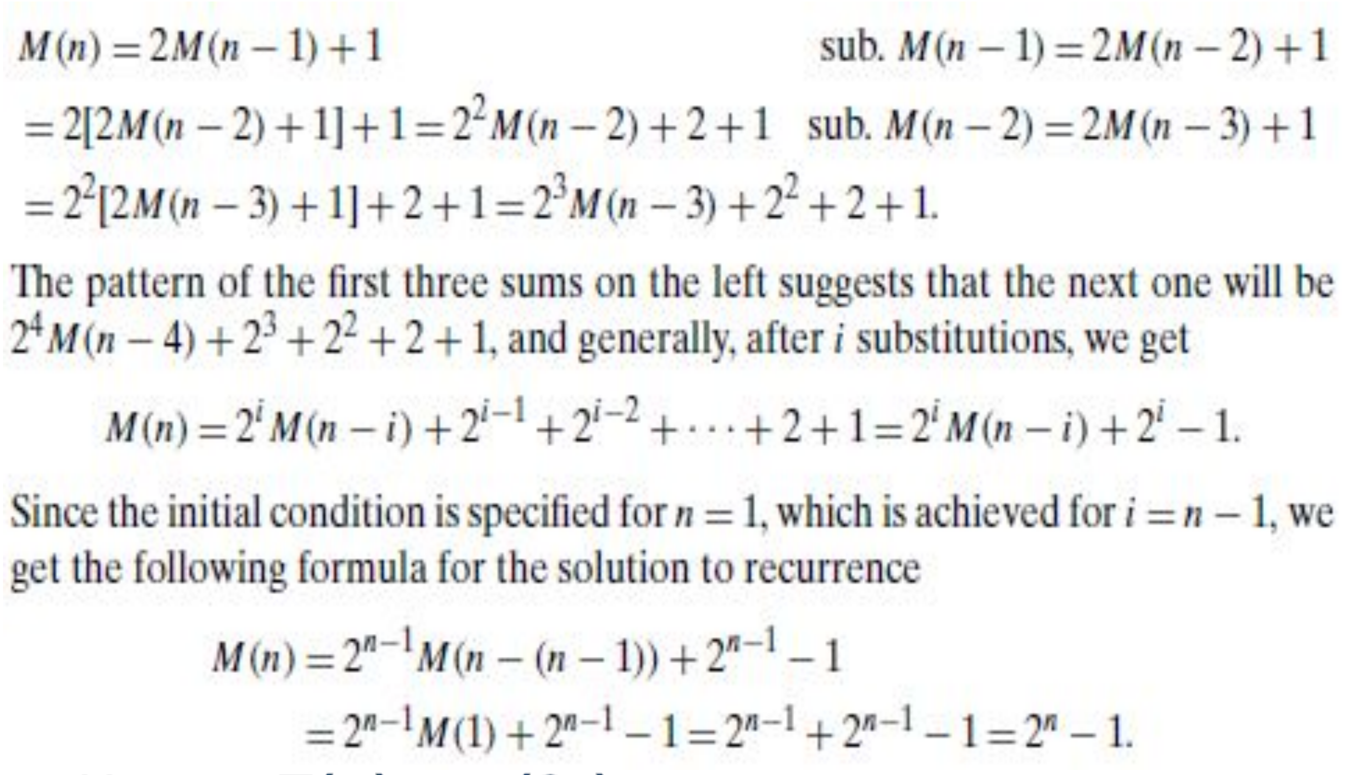
$M(1) = 1$ for $n=1$

Towers of Hanoi: Recursive

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

$M(1) = 1$ for n=1

Solving the above recurrence relation using the *method of **backward substitutions***.



$$M(n) = 2M(n-1) + 1 \qquad \text{sub. } M(n-1) = 2M(n-2) + 1$$

$$= 2[2M(n-2)+1] + 1 = 2^2M(n-2) + 2 + 1 \qquad \text{sub. } M(n-2) = 2M(n-3) + 1$$

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

The pattern of the first three sums on the left suggests that the next one will be $2^4M(n-4) + 2^3 + 2^2 + 2 + 1$, and generally, after i substitutions, we get

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

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence

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

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

$$\sum_{i=0}^{i=n} 2^i = 2^{n+1} - 1$$

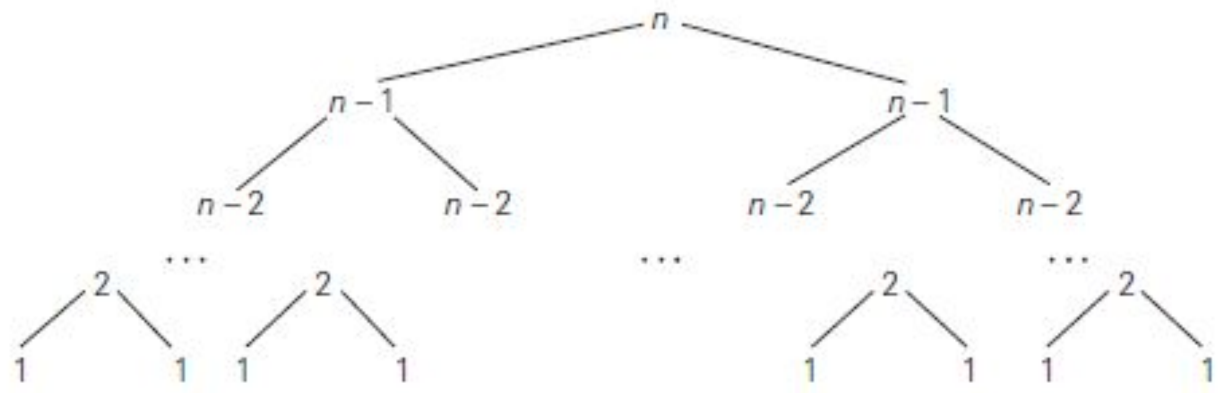
$M(n) = 2*2^{n-1} - 1$

$= 2*(2^n/2) - 1$

Hence, $T(n) = \Theta(2^n)$

Towers of Hanoi: Recursive

- Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.

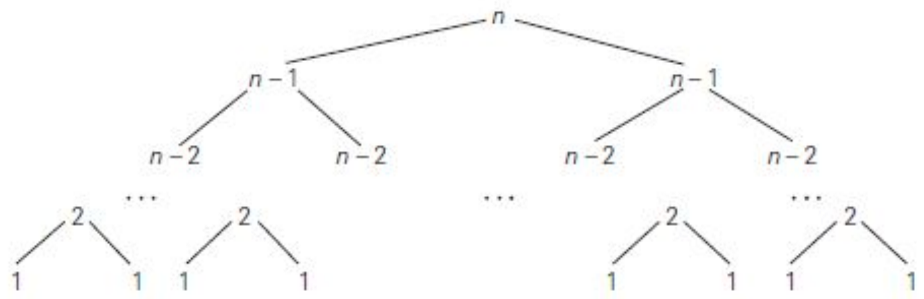


- When a recursive algorithm makes more than a single call to itself, it can be useful for analysis purposes to construct a tree of its recursive calls. In this tree, nodes correspond to recursive calls, and we can label them with the value of the parameter (or, more generally, parameters) of the calls. For the Tower of Hanoi example, the tree is given in Figure 2.5. By counting the number of nodes in the tree, we can get the total number of calls made by the Tower of Hanoi algorithm:

$$C(n) = \sum_{l=0}^{n-1} 2^l \text{ (where } l \text{ is the level in the tree in Figure } \quad) = 2^n - 1.$$

Towers of Hanoi: Recursive

- Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.



The number of nodes at level 0 = 2^0 = 1

The number of nodes at level 1 = 2^1 = 2

The number of nodes at level 2 = 2^2 = 4

………..

The number of nodes at level (n-1) = $2^{(n-1)}$

Total number of calls made by Towers of Hanoi:

$C(n)=2^0 + 2^1 + 2^2 + \ldots\ldots + 2^{(n-1)} = 2^n - 1$

Tower of Hanoi

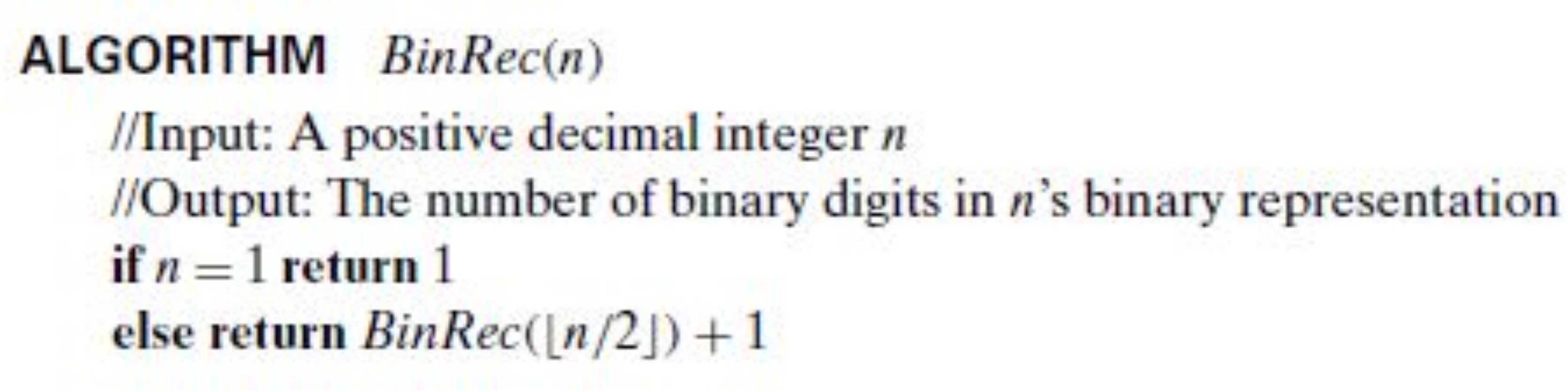
```c
/*C program for Tower of Hanoi using Recursion */
#include <stdio.h>

void towers(int, char, char, char);

int main()
{
int num;

printf("Enter the number of disks : ");
scanf("%d", &num);
printf("The sequence of moves involved in the Tower of Hanoi are :\n");
towers(num, 'A', 'C', 'B');
return 0;
}
void towers(int num, char frompeg, char topeg, char auxpeg)
{
if (num == 1)
{
printf("\n Move disk 1 from peg %c to peg %c", frompeg, topeg);
return;
}
towers(num - 1, frompeg, auxpeg, topeg);
printf("\n Move disk %d from peg %c to peg %c", num, frompeg, topeg);
towers(num - 1, auxpeg, topeg, frompeg);
}
```

Example 3:

- Counting bits in the binary representation of an decimal number

Counting bits in the binary representation of an decimal number

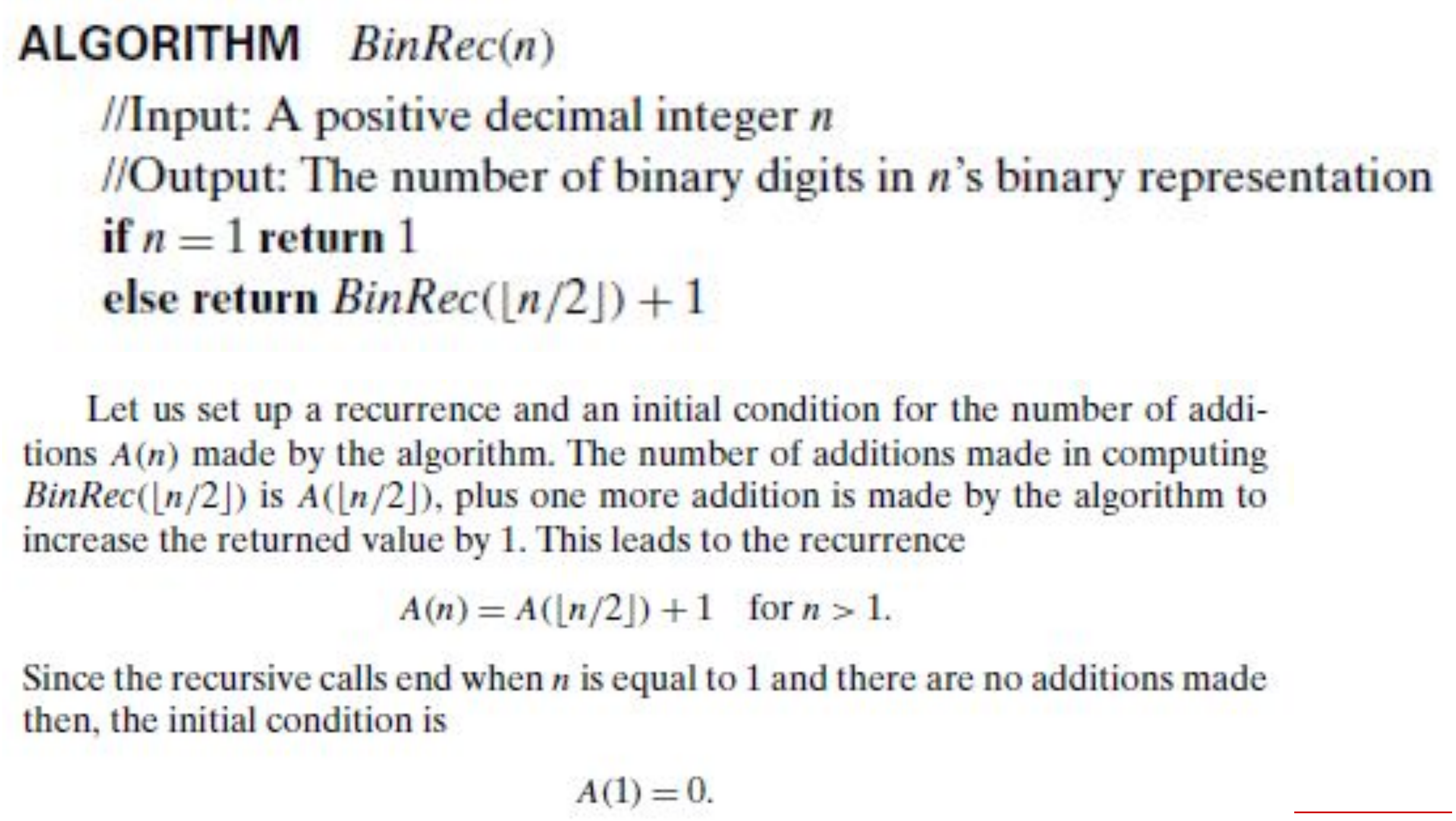
- Recursive Algorithms



```
ALGORITHM  $BinRec(n)$
    //Input: A positive decimal integer $n$
    //Output: The number of binary digits in $n$'s binary representation
    if $n = 1$ return 1
    else return $BinRec(\lfloor n/2 \rfloor) + 1$
```

Counting bits in the binary representation of an decimal number

- Recursive Algorithms



```
ALGORITHM BinRec(n)
    //Input: A positive decimal integer n
    //Output: The number of binary digits in n's binary representation
    if $n = 1$ return 1
    else return $BinRec(\lfloor n/2 \rfloor) + 1$
```

Let us set up a recurrence and an initial condition for the number of additions $A(n)$ made by the algorithm. The number of additions made in computing $BinRec(\lfloor n/2 \rfloor)$ is $A(\lfloor n/2 \rfloor)$, plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1.$$

Since the recursive calls end when n is equal to 1 and there are no additions made then, the initial condition is

$$A(1) = 0.$$

Solving the Recurrence Relation

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

Now backward substitutions encounter no problems:

$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots \\
&\dots \\
&= A(2^{k-i}) + i \\
&\dots \\
&= A(2^{k-k}) + k.
\end{aligned}$$

Thus, we end up with

$$A(2^k) = A(1) + k = k,$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$

Thanks for Listening

- Slides Courtesy: Dr Umadevi V
- Tower of Hanoi: https://in.video.search.yahoo.com/search/video;_ylt=AwrPiBRtFn1gBA0ADAa7HAx.;_ylu=Y29sbwNzZzMEcG9zAzEEdnRpZAMEc2VjA3BpdnM-?p=tower+of+hanoi+recursive+function+%2B+c+program&fr2=piv-web&fr=mcafee#id=3&vid=b2997b9bb4d3ce3af2ec40264bbf2d6c&action=view