Analysis and Design of Algorithms (23CS4PCADA)

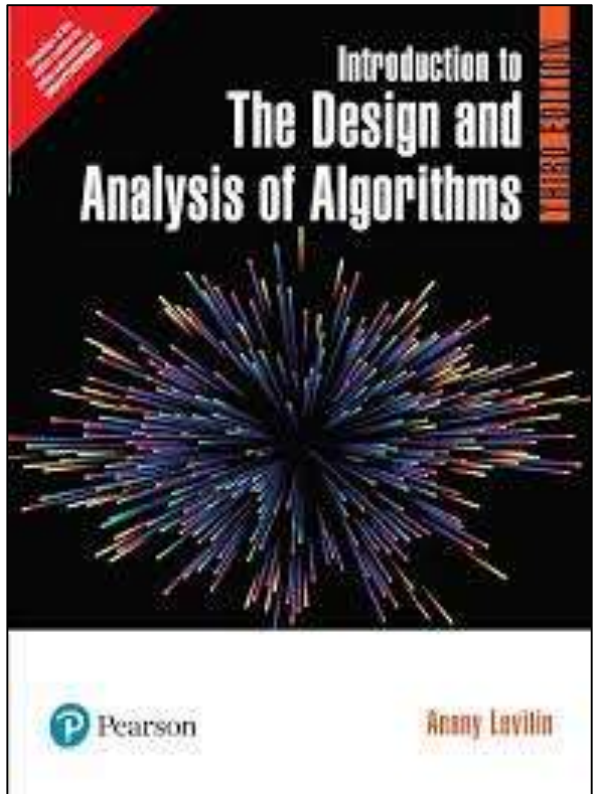
Course Instructor

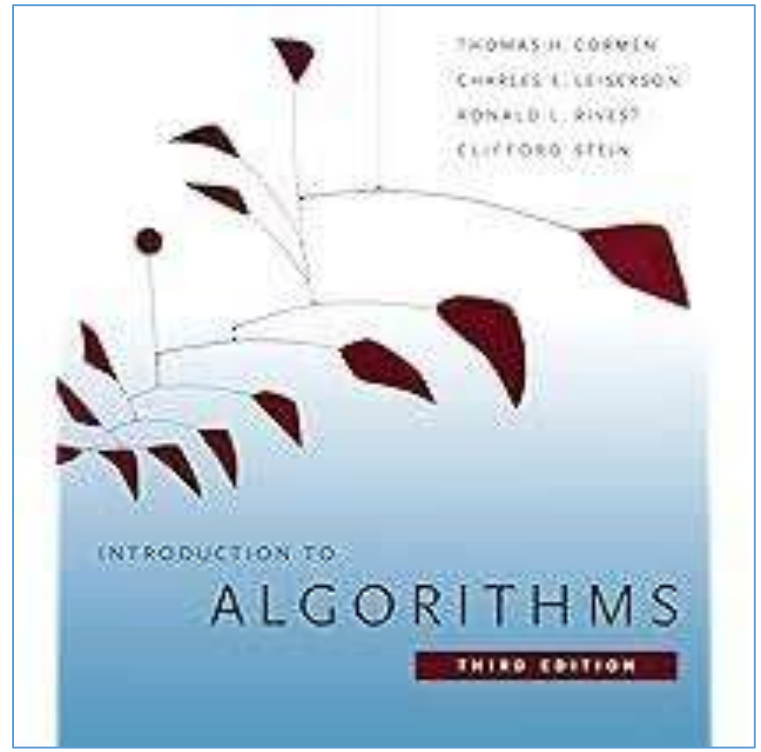
Dr. Kayarvizhy N, Professor, Dept of CSE

Kayarvizhyn.cse@bmsce.ac.in (9900502934)

Prescribed Text Book:

Sl.No.	Book Title	Authors	Edition	Publisher	Year
1	Introduction to the Design and Analysis of Algorithms	Anany Levitin	3rd Edition	Pearson	2014
2	Introduction to Algorithms	Thomas H Cormen, Charles E Leiserson,	Third	The MIT Press	2009





Assessment Plan (for 50 marks of CIE):

Tool	Remarks	Marks
Internals	TWO	20
QUIZ	ONE	5
Lab Component	CIE+ Two Lab Tests	25
Total		**50**

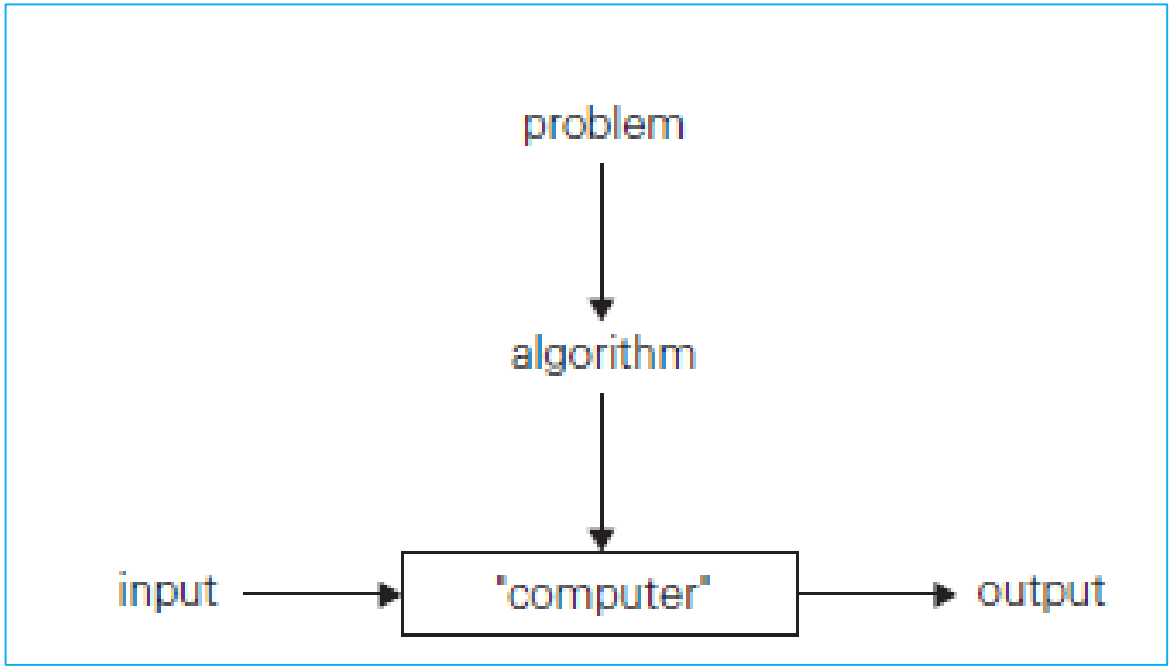
Laboratory Plan Instructions

- Design, develop and implement the specified algorithms for the following problems using any programming Language in LINUX/Windows environment, preferably using C language.
- For sorting and searching problems, the program should allow both manual entry of the array elements and also reading of array elements using random number generator. Plot a graph of the time taken versus N using MS Excel and paste the same in the record.
- Observation book to be maintained for Continuous Internal Evaluation
- Lab Record–Soft copy of the record.
- For the first three lab session the students are to be introduced on the Hacker rank/ Leetcode platform to solve problems (eg: Tower of Hanoi, Linear Search, Binary Search, Bubble Sort, Selection Sort, Insertion Sort, etc.)

Introduction to Algorithm

An Algorithm is a sequence of unambiguous instructions for solving a problem for any legitimate input in a finite amount of time

- The **nonambiguity requirement** for each step of an algorithm cannot be compromised.
- The **range of inputs** for which an algorithm works has to be specified carefully.
- The **same algorithm can be represented in several different ways**.
- There may exist **several algorithms for solving the same problem**.



The notion of the algorithm.

Algorithms for the **same problem** can be based on very different ideas and can solve the problem with dramatically different speeds.

Greatest Common Divisor

Greatest Common Divisor of two nonnegative, not-both-zero integers m and n, denoted gcd(m, n), is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero

Euclid's algorithm

- **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*

gcd(60, 24) = gcd(24, 12) = gcd(12, 0) = 12.

How do we know that Euclid's algorithm eventually comes to a stop?

Consecutive integer checking algorithm for computing gcd*(m, n)*

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

Note that unlike Euclid's algorithm, this algorithm, in the form presented, does not work correctly when one of its input numbers is zero. This example illustrates why it is so important to specify the set of an algorithm's inputs explicitly and carefully.

Middle-school procedure for computing gcd(m, n)

Step 1 Find the prime factors of m.

Step 2 Find the prime factors of n.

Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If p is a common factor occurring p_m and p_n times in m and n, respectively, it should be repeated $\min\{p_m, p_n\}$ times.)

Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

gcd(60, 24)

60 = 2 . 2 . 3 . 5
24 = 2 . 2 . 2 . 3
gcd(60, 24) = 2 . 2 . 3 = 12.

- The middle school procedure does not qualify, in the form presented, as a legitimate algorithm. Why? Because the prime factorization steps are not defined unambiguously:
- Step 3 is also not defined clearly enough. How would you find common elements in two sorted lists?

Fundamentals of Algorithmic Problem Solving

- Understanding the problem
- Ascertaining the capabilities of the Computational Device
- Choosing between Exact and Approximate Problem Solving
- Algorithm Design Techniques
- Designing an Algorithm and Data Structures
- Methods of Specifying an algorithm
- Proving an Algorithm’s Correctness
- Analyzing an algorithm
- Coding an algorithm

Understanding the problem

- The problem given should be understood properly
  - Ask questions to clarify any doubts, work through examples manually, consider special cases, and seek further clarification if necessary.
- Clearly specify the set of instances the algorithm needs to handle.

- Correct Algorithm should work for all possible inputs
  - Boundary Conditions

Ascertaining the capabilities of the Computational Device

- To know the capabilities of the computational device where the algorithm will run.
  - The type of architecture
  - Speed
  - Memory Availability
- The algorithm is designed as Sequential / Parallel Algorithm depends on Computational device

Exact/Approximate Solution

- Once algorithm is devised, it is necessary to show that it compute answer for all the possible inputs

- The solution is stated in two forms
  - Solving the problem exactly
  - Solving it approximately
    - Example : Finding Square root of number, solving nonlinear equations

Algorithm Design Techniques

- An ***algorithm design technique*** (or “strategy” or “paradigm”) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
- Some important design techniques are
  - Brute Force
  - Greed Method
  - Divide and Conquer
  - Dynamic Programming
  - Backtracking
  - Linear Programming

Designing an Algorithm and Data Structures

- Data structure remain important for both design and analysis of algorithms
- Data structures can be defined as a particular scheme of organizing related data items
- Some of the elementary data structures are Array, List, Set

Algorithm + Data Structure = Programs

Methods of Specifying the Algorithm

- These are the two options that are most widely used nowadays for specifying algorithms.
  - ***Pseudocode***
  - ***flowchart***

Proving an Algorithm's Correctness

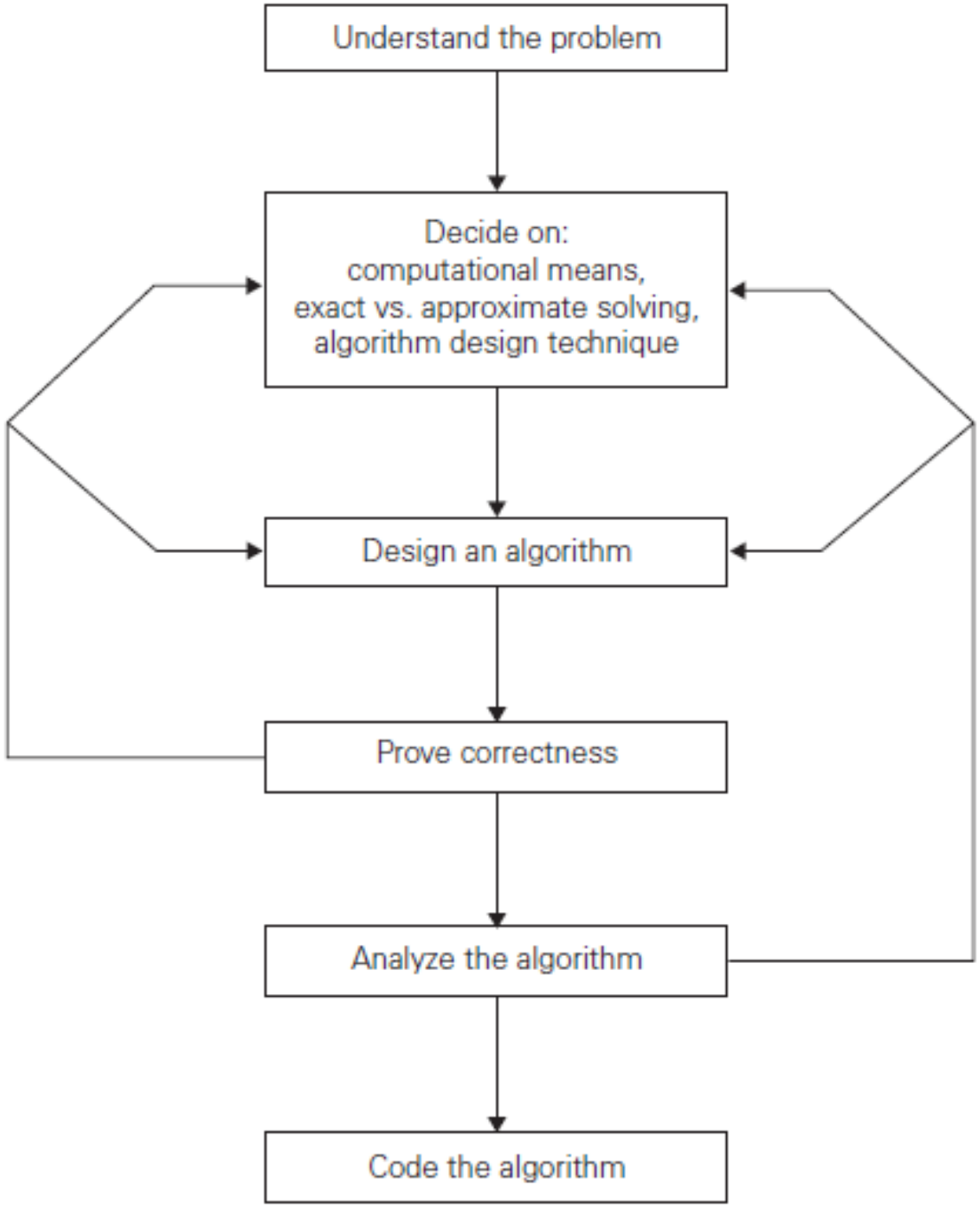
- Once an algorithm has been specified, you have to prove its ***correctness***
- To prove that the algorithm yields a required result for every legitimate input in a finite amount of time.
  - For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd*(m, n)* = gcd*(n, m* mod *n)*
- A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs

Analyzing an Algorithm

- The algorithm is analyzed by its efficiency
- There are two kinds of algorithm efficiency
  - ***Time efficiency*** : indicating how fast the algorithm runs
  - ***Space efficiency***: indicating how much extra memory it uses

Coding an Algorithm

- Programming an algorithm presents both a peril and an opportunity.
- The peril lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently.
- The validity of programs is still established by testing



Algorithm design and analysis process

Analysis of Algorithm Efficiency – Analysis Framework

- Measuring an Input's Size
- Units for Measuring Running Time
- Orders of Growth
- Worst-Case, Best-Case, and Average-Case Efficiencies

Measuring an Input's Size

- Time required by an algorithm is proportional to size of the problem instance.
- For e.g., more time is required to sort 20 elements than what is required to sort 10 elements.

Units for Measuring Running Time

- Count the number of times an algorithm's basic operation is executed. (Basic operation: The most important operation of the algorithm, the operation contributing the most to the total running time.)

ALGORITHM sum_of_numbers (A[0… n-1])
// Functionality : Finds the Sum
// Input : Array of n numbers
// Output : Sum of 'n' numbers
i ← 0
sum ← 0
while i < n
    sum ← sum + A[i] ⟶ n
    i ← i + 1
return sum

Total number of steps for basic operation execution, C (n) = n

Orders of Growth

- The order of growth of an algorithm is an approximation of the time required to run a computer program as the input size increases
- For example, a program with a linear order of growth generally requires double the time if the input doubles.

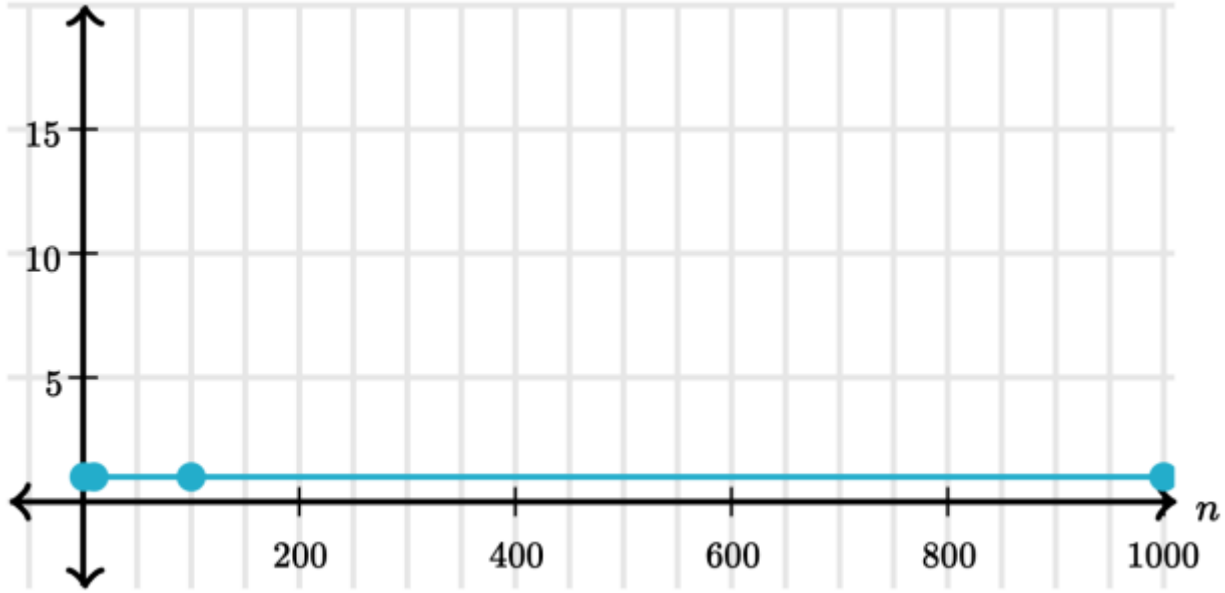
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Constant time

- When an algorithm has a constant order of growth, it means that it always takes a fixed number of steps, no matter how large the input size increases.

```
first_post = posts[0]
```

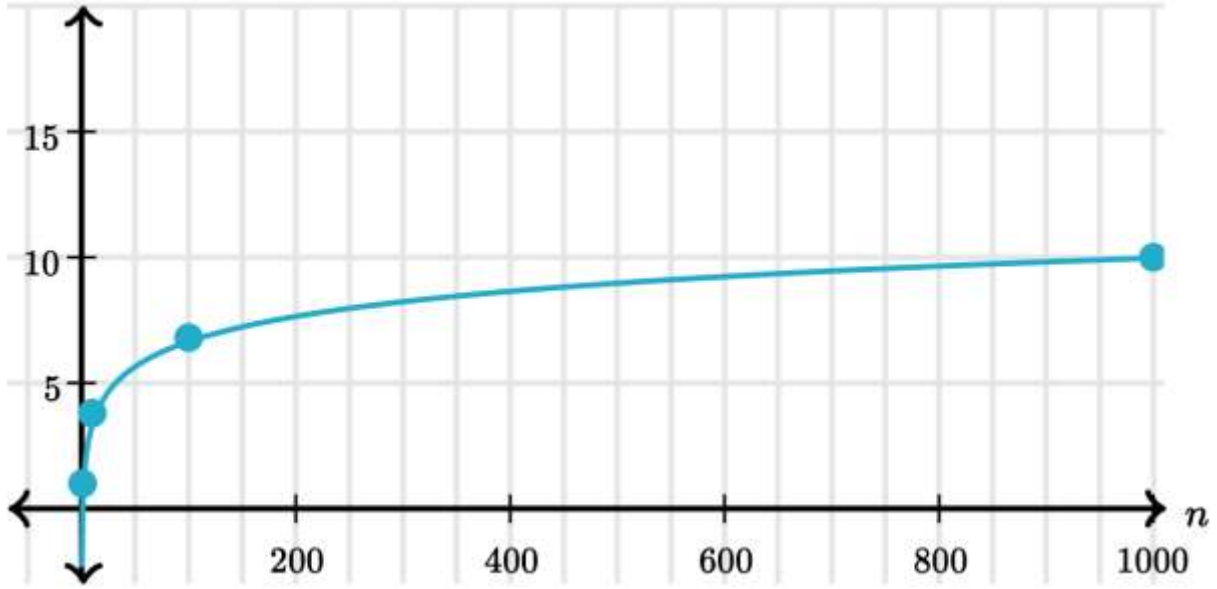
List size	Steps
1	1
10	1
100	1
1000	1



Logarithmic time

- When an algorithm has a logarithmic order of growth, it increases proportionally to the logarithm of the input size.
- The binary search algorithm is an example of an algorithm that runs in logarithmic time.

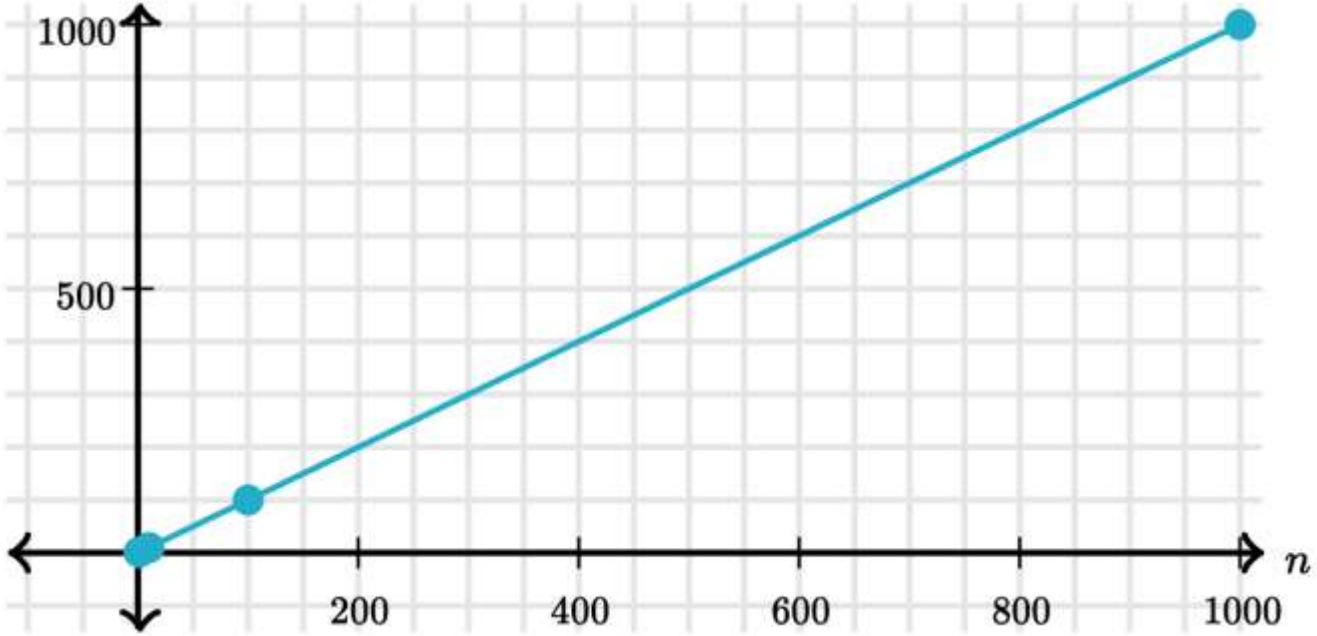
List size	Steps
1	1
10	4
100	7
1000	10



Linear time

- When an algorithm has a linear order of growth, its number of steps increases in direct proportion to the input size.
- Linear search algorithm runs in linear time.

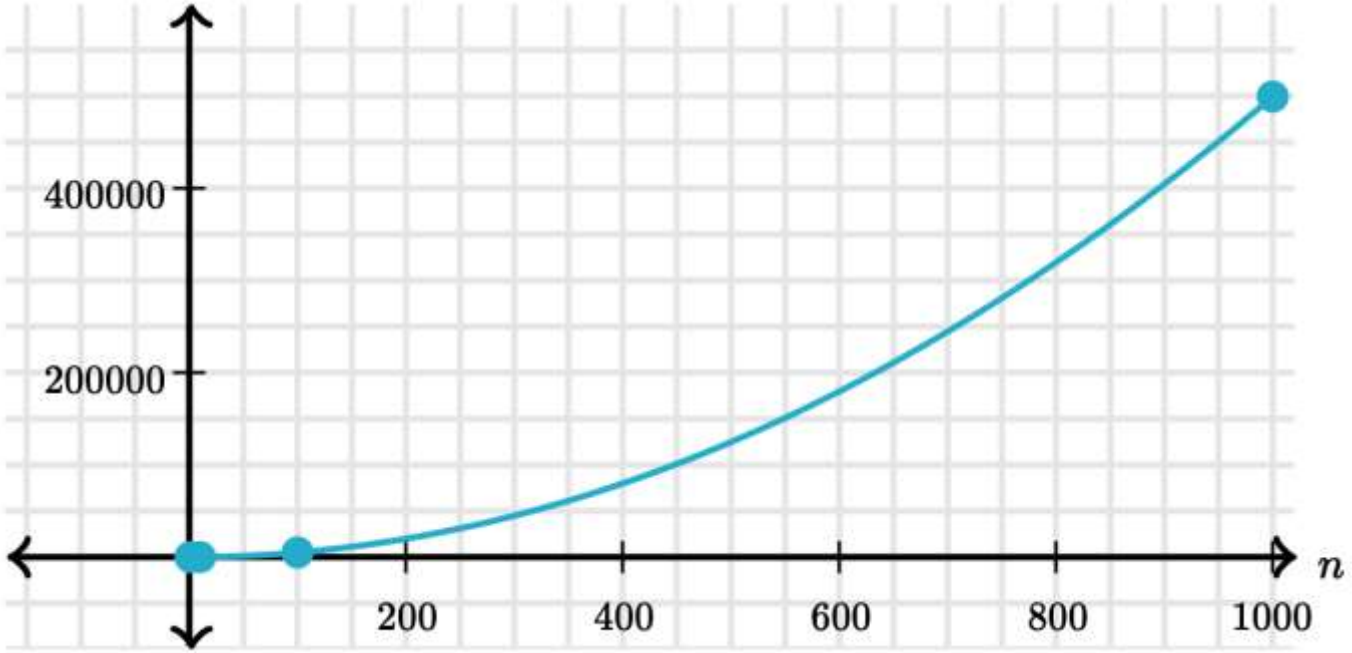
List size	Steps
1	1
10	10
100	100
1000	1000



Quadratic time

- When an algorithm has a quadratic order of growth, its steps increase in proportion to the input size squared.
- **Selection sort** - algorithm starts from the front of the list, then keeps finding the next smallest value in the list and swapping it with the current value.

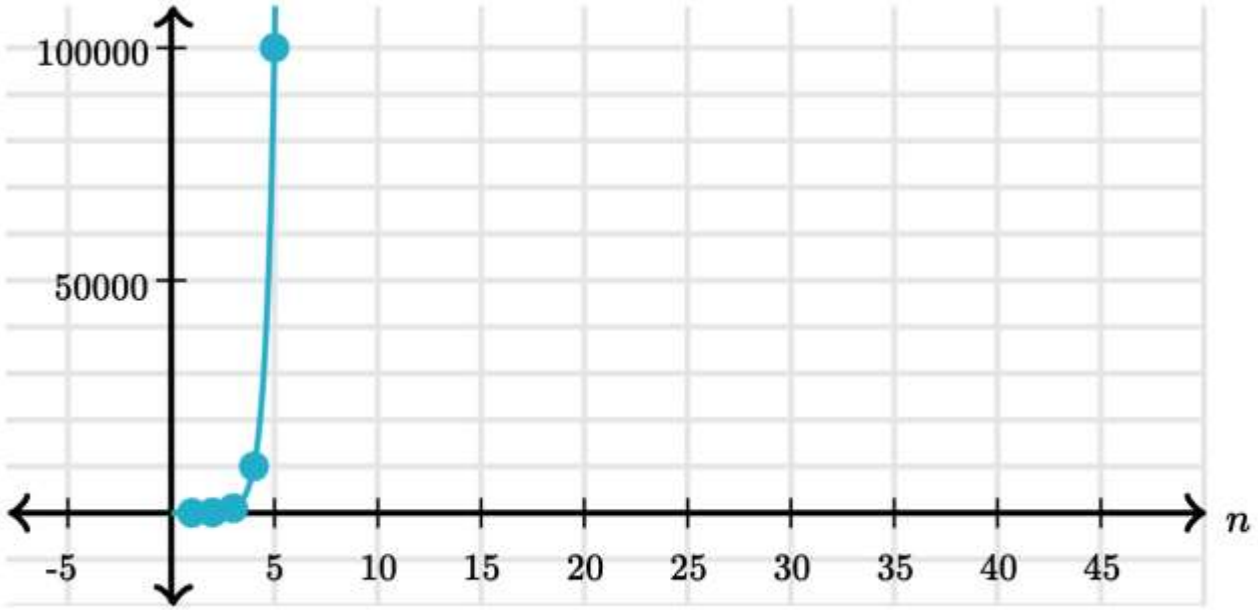
List size	Steps
1	1
10	45
100	4950
1000	499500



Exponential time

- When an algorithm has a superpolynomial order of growth, its number of steps increases faster than a polynomial function of the input size.
- An algorithm that generates all possible numerical passwords for a given password length.

Digits	Steps
1	10
2	100
3	1000
4	10000
5	100000



Worst-Case, Best-Case, and Average-Case Efficiencies

Algorithm efficiency depends on the input size n and for some algorithms efficiency depends on type of input. We have best, worst & average case efficiencies.

KINDS OF ANALYSIS

Worst-case: (usually)

- T(n) = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- T(n) = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case:

- T(n) = minimum time of algorithm on any input of size n.

Worst-case efficiency

- The ***worst-case efficiency*** of an algorithm is its efficiency for the worst-case input of size *n*, which is an input (or inputs) of size *n* for which the algorithm runs the longest among all possible inputs of that size.
- What kind of inputs yield the largest value of the basic operation's count C(n) among all possible inputs of size n

Example : Sequential Search

Worst Case Input: Search element is at the last or not found

Best-case efficiency

- The ***best-case efficiency*** of an algorithm is its efficiency for the best-case input of size *n*, which is an input (or inputs) of size *n* for which the algorithm runs the fastest among all possible inputs of that size.

- Inputs for which the count C(n) will be the smallest among all possible inputs of size n.

- The best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.

- For example, the best-case inputs for sequential search are lists of size n with their first element equal to a search key

Average-case efficiency

- Neither the worst-case analysis nor its best-case counterpart yields the necessary information about an algorithm's behavior on a "typical" or "random" input

- This is the information that the ***average-case efficiency*** seeks to provide

- To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size *n*.

Assumptions (Sequential Search)

- The probability of a successful search is equal to p $(0 \le p \le 1)$
- The probability of the first match occurring in the ith position of the list is the same for every i.
- **Average number of key comparisons $C_{avg}(n)$ as**

$$
\begin{aligned}
C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p) \\
&= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p) \\
&= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).
\end{aligned}
$$

- In the case of a successful search, the probability of the first match occurring in the ith position of the list is p/n for every i, and
- the number of comparisons made by the algorithm in such a situation is obviously i.
- In the case of an unsuccessful search, the number of comparisons will be n with the probability of such a search being $(1- p)$.

Asymptotic Notations and Basic Efficiency Classes

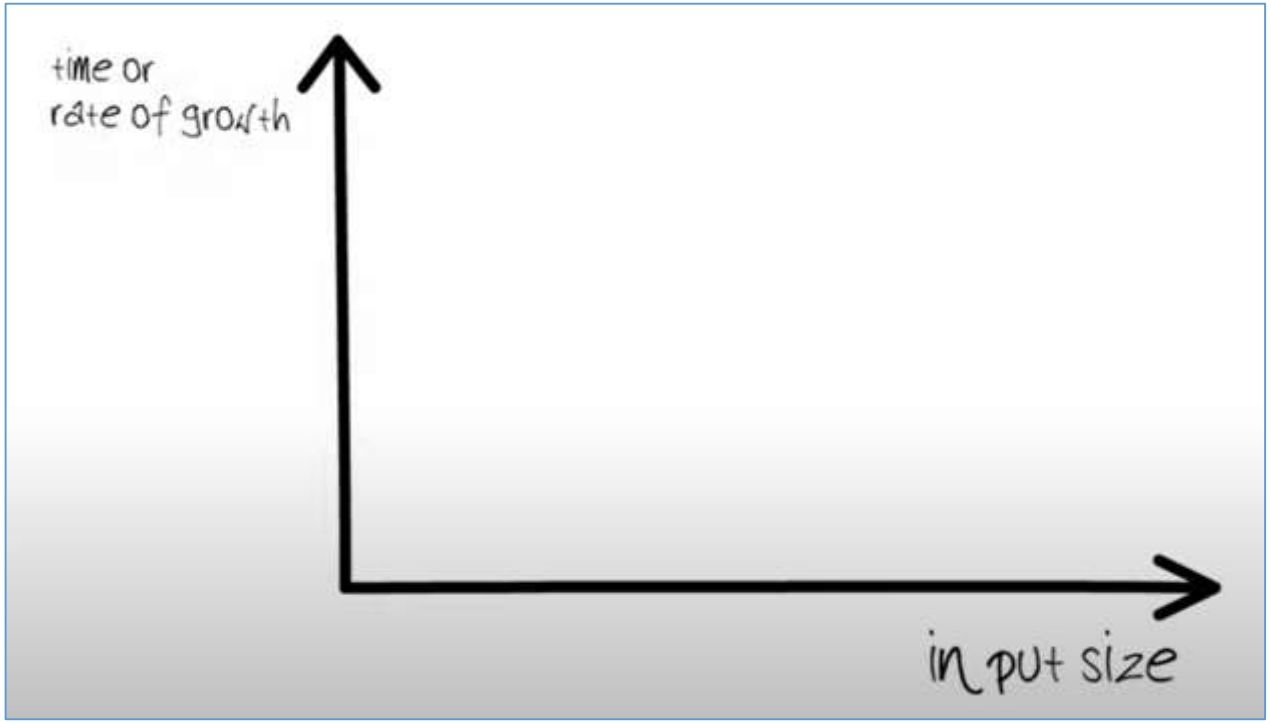
- Asymptotic Notations are mathematical tools **used to analyze the performance of algorithms by understanding how their efficiency changes as the input size grows.**
- These notations provide a **concise way to express the behavior of an algorithm's** time or space complexity as the input size approaches infinity.
- Rather than **comparing algorithms** directly, asymptotic analysis focuses on understanding the relative growth rates of algorithms' complexities.
- Asymptotic analysis allows for the comparison of algorithms' space and time complexities by examining their performance characteristics as the input size varies

O- "Big-Oh" notation

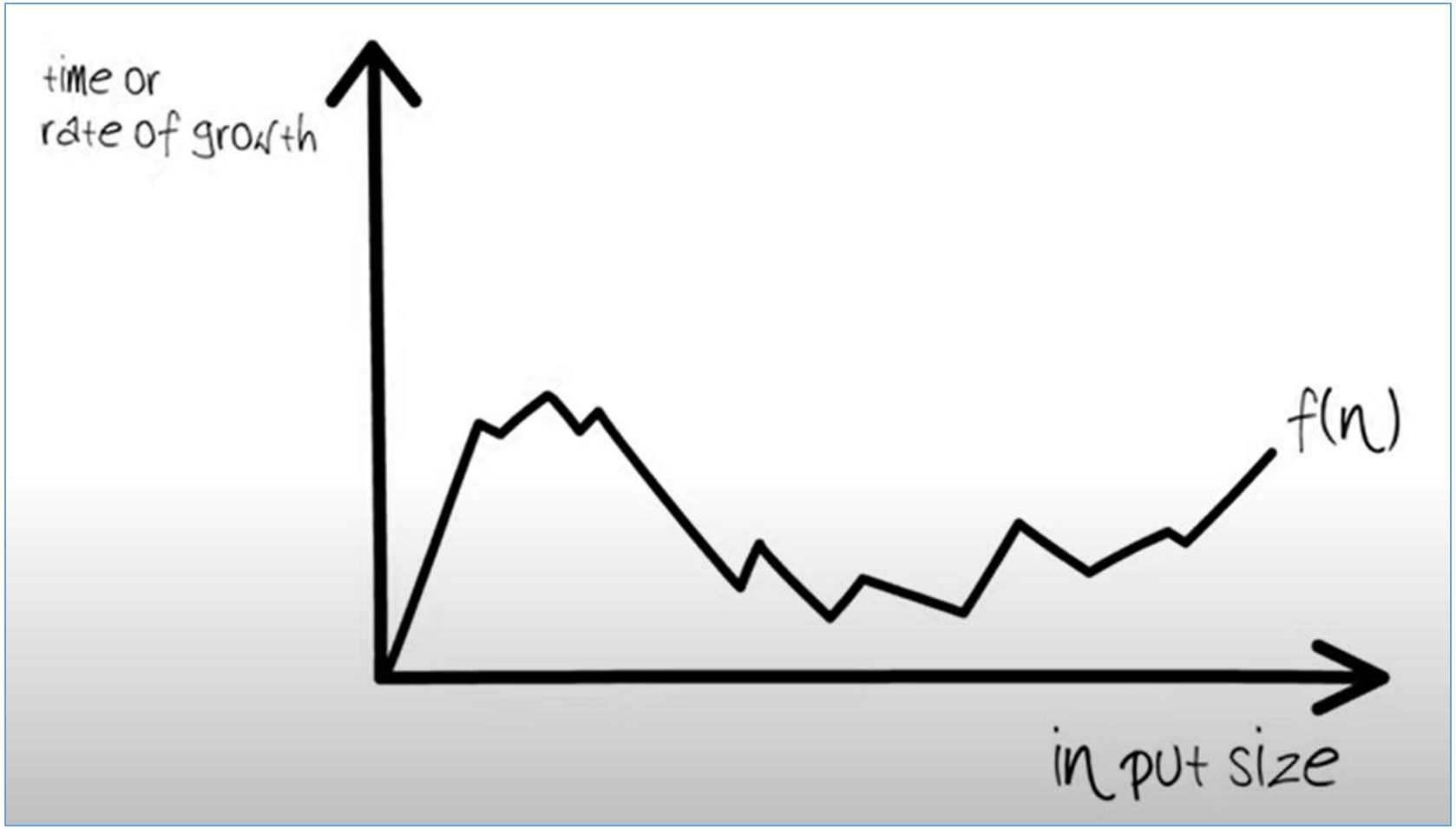
- It describes the upper bound of an algorithm's running time.
- It measures the worst-case time complexity or the maximum or longest amount of time of an algorithm can possibly take to complete

O- “Big-Oh” notation (Upper Bound)

- f(n) is the function which represents our Algorithm

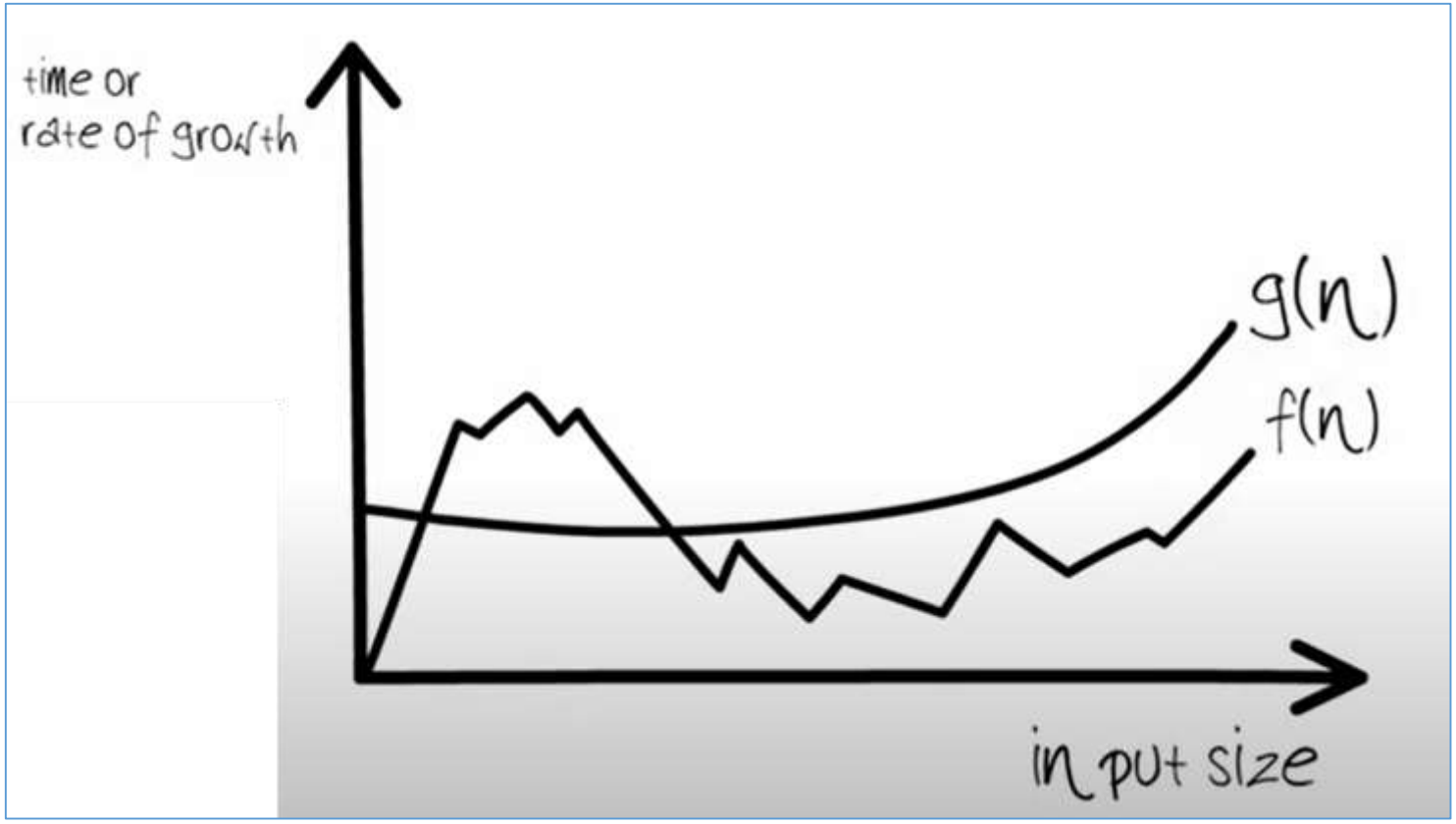


O- "Big-Oh" notation



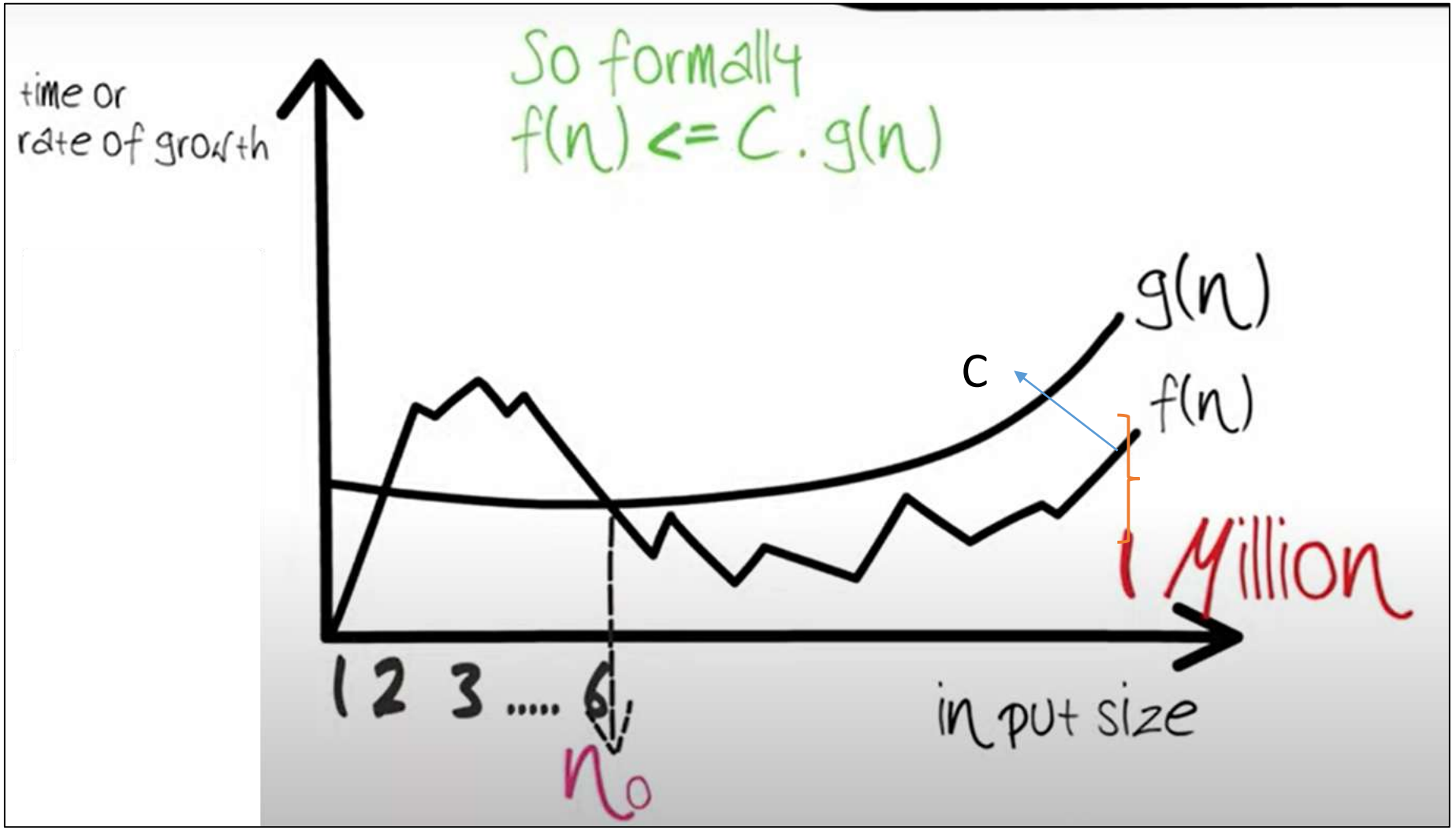
The graph represents the growth of the function f(n)

O- "Big-Oh" notation



- Aim is to find the worst case complexity of an algorithm
  - Have to find the upper bound
  - Need to find a function which slightly greater than our f(n) function
  - Lets call that function as g(n)

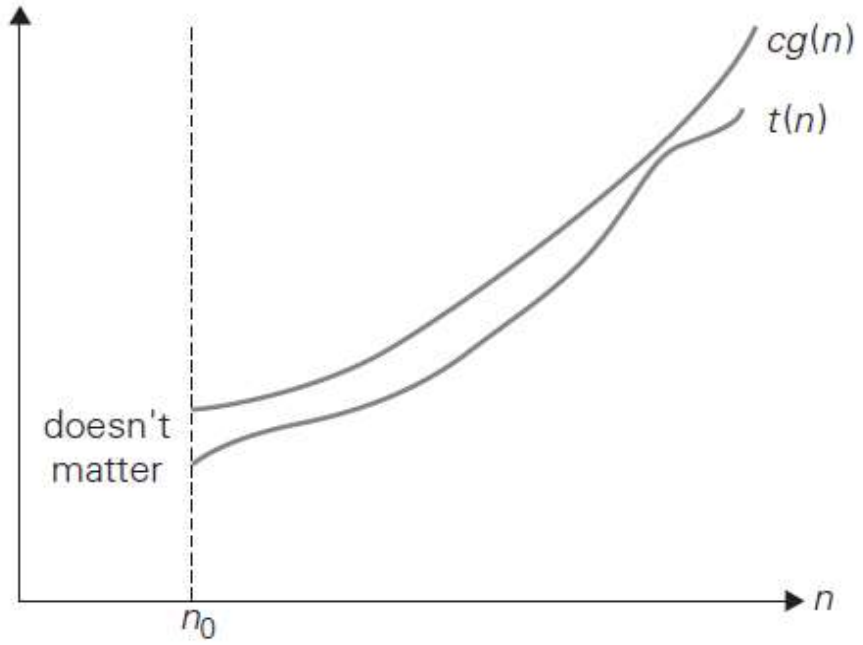
O- "Big-Oh" notation



- F(n)=O(g(n) (Means c.g(n) is the upper bound of the function)
- N_0 is the threshold of the function
- C is the Constant

O- "Big-Oh" notation

- A function *t (n)* is said to be in *O(g(n))*, denoted *t (n)* ∈ *O(g(n))*, if *t (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



Big-oh notation: $t(n) \in O(g(n))$.

$t(n) \leq cg(n)$ for all $n \geq n_0$.

O- "Big-Oh" notation

Find upper bound for $f(n) = n^2 + 1$

$f(n) <= c.g(n)$

$n^2 + 1 <= c.n^2$

$n^2 + 1 <= 2.n^2$,for n >= 1

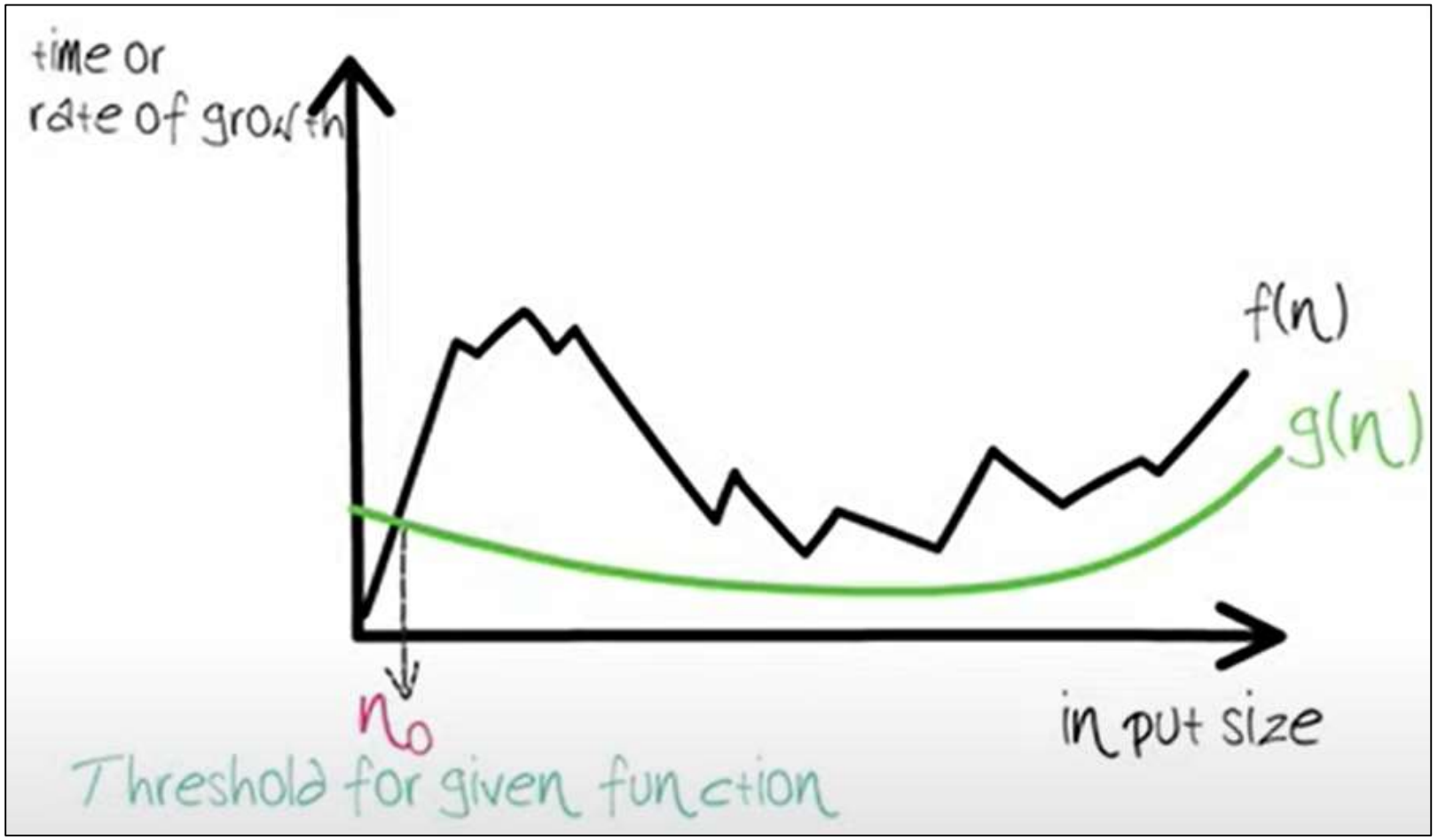
$$n^2 + 1 = O(n^2)$$ for c = 2 and n0 >= 1

Ω Big Omega Notation

- It describes the lower bound of an algorithm's running time

- It measures the best-case time complexity or the minimum or shortest amount of time an algorithm can possibly take to complete

Ω Big Omega Notation
(Lower Bound Function)

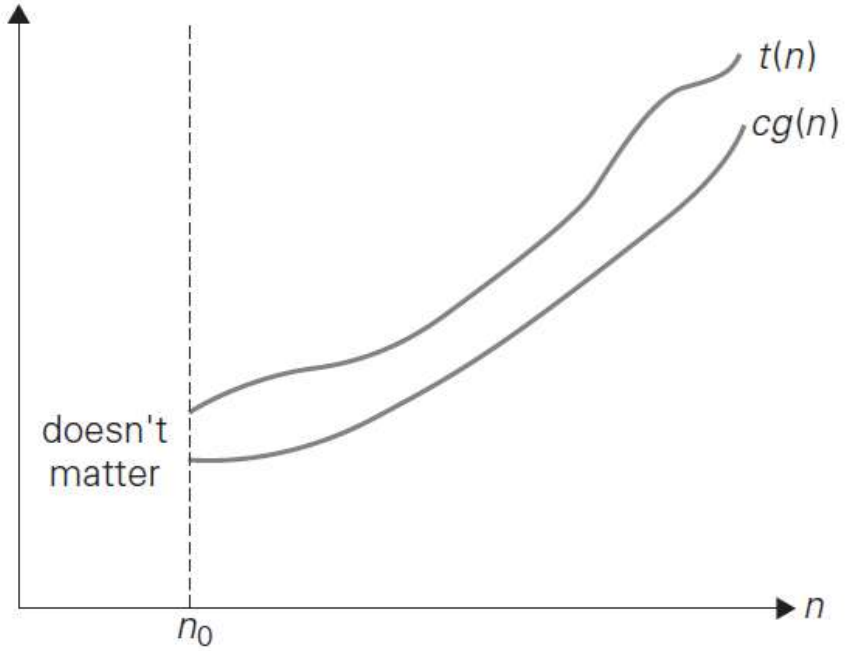


So formally
$f(n) >= C . g(n)$

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

Ω Big Omega Notation

- A function *t (n)* is said to be in Ω*(g(n)),* denoted *t (n)* ∈ *(g(n)),* if *t (n)* is bounded below by some positive constant multiple of *g(n)* for all large *n,* if there exist some positive constant *c* and some nonnegative integer *n*0 such that



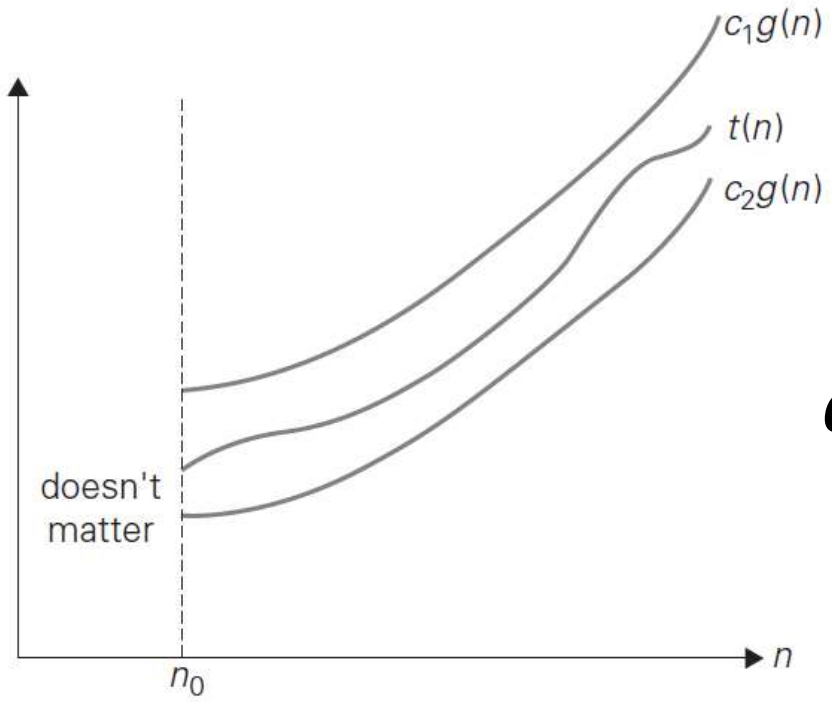
t (n) ≥ *cg(n)* for all *n* ≥ *n*0.

Θ Big Theta- Notation

- It describes the average case running time of an algorithm.
- It is the tightest bound on the running time of an algorithm

Θ Big Theta- Notation

- A function *t (n)* is said to be in Θ*(g(n))*, denoted *t (n)* ∈ *(g(n)),* if *t (n)* is bounded both above and below by some positive constant multiples of *g(n)* for all large *n,* i.e., if there exist some positive constants *c*1 and *c*2 and some nonnegative integer *n*0 such that



$c_2g(n) \leq t\,(n) \leq c_1g(n)$ for all $n \geq n_0$.

Prove that $f(n) = \Theta\, g(n)$ where

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

Lowerbound: $c1.\ g(n) <= f(n)$

$c1.\ n <= 3n + 2$

$c1 = 1$ and $n >= 1$

$n <= 3n+2$

for $n>=1$ is True

Upperbound: $f(n) <= c2.\ g(n)$

$3n + 2 <= c2\ .\ n$

for $c2 = 4$ and $n>=1$

$3n + 2 <= 4n$

for $n>=1$ is True

$3n+2 = \Theta\,(n)$

Some Common Algo's Complexities

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 2.4: Growth rates of common functions measured in nanoseconds

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

Basic Efficiency classes

Mathematical Analysis of Nonrecursive Algorithms

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count or, at the very least, establish its order of growth

Mathematical Analysis of Recursive Algorithms

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

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
```

i/p size : n
A[i] = A[j] : Basic op

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

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

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

$n-1-[n-2]$
$n-1-(n-2)$

$$= (n-1) + (n-2) + \cdots + 1$$

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

$$= \frac{n^2}{2} - \frac{n}{2}$$

$$C_{worst}(n) = n^2$$

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
```

i/p size: no of elements
maxval ← A[i] :→ Basic op

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 C(n):

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

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

$$C(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n) \quad \{\text{Average case}\}$$

(or)

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

MatrixMultiplication(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])
// Multiplies 2 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

i/p : matrix order n
mul & add : Basic op

Total number of multiplication M(n) is expressed by the following triple sum

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

Starting with the innermost sum $\sum_{k=0}^{n-1} 1$, which is equal to n. We get

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = \boxed{n^3}$$

```
Algorithm Mystery(n)
{
    S = 0;
    for i = 1 to n do
    { S = S + i * i; }
    return S
}
```

1) Find what does this algorithm Compute?

The algorithm Computes the sum of Squares of numbers from 1 to n.

2) What is its basic operation?

The basic operations are multiplication and addition

3) How many times is the basic operation executed?

$$\sum_{i=1}^{n} 1 = n$$

4) What is the efficiency class of this algorithm

Since "for loop" executes once for each of the numbers from 1 to n, so this shows that for loop executes n times. The basic operation is best, worst or average case runs n times. Hence the Running time of $\Theta(n)$.

①
```
int a=0, b=0
for(i=0; i<N; i++)
{
    for(j=0; j<N, j++)
    {
        a=a+j;
    }
}
for(k=0; k<N; k++)
{
    b=b+k
}
```

$$\rightarrow C(N) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 1$$

$$= \sum_{i=0}^{N-1} (N-1-0+1)$$

$$= \left(\sum_{i=0}^{N-1} \right) N$$

This also executes N times.

$$= N * N = N^2$$

$O(N^2)$

$\sum_{k=0}^{N-1} 1$ ← { for(k=0; k<N; k++) b=b+k

= N times

O(N)

Since this program depends only on only variable N. you should choose higher order as your time complexity.

---

②
```
if (x<7)
    print "x is lesser than 7"
else
{
    for(i=1; i<=n; i++)
        print i;
}
```

"if" part executes in constant time O(1)

Else part executes n times

When a program has if else part, we can't predict which part of the code executes. So choose whichever part of the code gives higher order time complexity

(3)

```
for(x=1; x<=n; x*=2)
    print x;
```

The above loop executes as follows.

Example:

n = 10

x value	count
1	1
2	1
4	1
8	1

} when i/p size is 10, loop executes 4 times, because x increments as multiple of two.

i/p	10	13	15	16	18
count	4	4	4	5	5

The above table demonstrates that, by change in input size, loop run for $O(\log n)$ steps.

(4)

```
int a=0, b=0;
for (i=0; i<N; i++)
{
    a = a + rand();
}
for (j=0; j<M; j++)
{
    b = b + rand();
}
```

$$C(n) = \sum_{i=0}^{N} 1 = N$$

$$C(n) = \sum_{j=0}^{M} 1 = M$$

$$O(N+M)$$

---

(5)

```
int a=0
for (i=0; i<N; i++)
{
    for (j=N; j>i; j--)
    {
        a = a+i+j;
    }
}
```

$$C(n) = \sum_{i=0}^{N-1} \sum_{j=N}^{i+1} 1 = \sum_{i=0}^{N} N-i$$

$$= N + (N-1) + (N-2) + \dots + 1 + 0$$

$$= \frac{N * (N+1)}{2}$$

$$= O(N^2)$$

(6)

```
int i, j, k = 0
for (i = n/2; i <= n; i++)
{
    for (j = 2; j <= n; j = j * 2)
    {
        k = k + n/2;
    }
}
```

j keeps doubling till it is less than or equal to n.

for n = 16, j = 2, 4, 8, 16

n = 32, j = 2, 4, 8, 16, 32.

So, j would run for O(log n) steps.

i runs for n/2 steps.

So, total steps = $O(n/2 * \log(n)) = O(n * \log n)$

What are Recurrence Relations.

→ It is an equation that defines a sequence of values.

→ Future terms derived as function of previous term.

Factorial

```
Factorial(n)
{
    if n=0
        return 1
    else
        return n * Factorial(n-1)
}
```

Base case $n = 0$
$T(0) = 1$

Basic operation
$n * Factorial(n-1)$

$$T(n) = T(n-1) + 3 \quad \text{if } n > 0$$

$$= T(n-2) + 6$$

$$= T(n-3) + 9$$

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

Assume $K = n$, $n - K = 0$

$$= T(0) + 3n$$

$$= 3n + 1$$

$$O(n)$$

Algorithm TowerofHanoi (n, S, D, T)

// Solve the Tower of Hanoi problem

// Input: no of disc's = n through poles S, D, T

// output: All n disc's stacked on D

if (n == 1)
    move a disc from S to D

Base case: if n=1 we do ONE movement.

else
    TowerofHanoi (n-1, S, T, D)
    move n^{th} disc from S to D
    TowerofHanoi (n-1, T, D, S)

Basic Operation: Movement of disc (M)

$T(1) = 1$

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

$= 2T(n-1) + 1$ — (1)

$n = n-1$ in (1)

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

$= 2T(n-2) + 1$ — (2)

Sub (2) in (1)

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

$= 4T(n-2) + 2 + 1$ — (3)

$n = n-2$ in (1)

$T(n-2) = 2T(n-3) + 1$ — (4)

Sub (4) in (3)

$= 4(2T(n-3) + 1) + 2 + 1$

$= 8T(n-3) + 4 + 2 + 1$

$2^3 T(n-3) + 2^2 + 2^1 + 2^0$

General formula

$$2^k T(n-k) + 2^{k-1} + 2^{k-2} + \dots + 2^1 + 2^0$$

$$n - k = 1$$
$$k = n - 1$$

$$2^{n-1} T(1) + 2^{n-2} + 2^{n-3} + \dots + 2^0$$

$$2^{n-1} + 2^{n-2} + 2^{n-3} + \dots + 2^1 + 2^0$$

$$T(n) = 2^{n-1}$$

$$O(2^n)$$

Solving Recurrence relations

(1) Solve the following recurrence relation

$x(n) = x(n-1) + 5$ for $n > 1$ and $x(1) = 0$

Sol

The above recurrence relation can be written as,

$$x(n) = \begin{cases} x(n-1)+5 & , \text{if } n>1 \\ 0 & , \text{if } n=1 \end{cases}$$

$x(n) = x(n-1) + 5$

Using backward substitution method, substitute $n-1$ in the place of n

$= [x(n-1-1) + 5] + 5$

$= x(n-2) + 2(5)$

$= [x(n-1-2) + 5] + 2(5)$

$= x(n-3) + 3(5)$ — problem reduces to $n-1$

$\vdots$

$= x(n-(n-1)) + (n-1)5$

$= x(n-n+1) + (n-1)5$

$= x(1) + (n-1)5$ ← Base condition

$= 0 + (n-1)5$

$= 5.(n-1)$

(2) Solve the following recurrence relation.

$x(n) = 3x(n-1)$ for $n \geq 1$, $x(1) = 4$

Sol

The recurrence relation can be written as

$$x(n) = \begin{cases} 4, & \text{if } n = 1 \\ 3x(n-1) & \text{if } n > 1 \end{cases}$$

$x(n) = 3x(n-1)$

~~$x(n-1) = 3[3x(n-2-1)]$~~

$x(n-1) = 3[3x(n-1-1)]$

$= 3^2 x(n-2)$ → Problem reduces to $n-1$

$= 3^3 x(n-3)$

$\vdots$

$= 3^{n-1} x(n-(n-1))$

$= 3^{n-1} x(1)$ Substitute $x(1) = 4$

$= 3^{n-1} \cdot 4$.

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

$T(n) = n * T(n-1)$ ——— (1)

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

$= (n-1) * T(n-2)$ ——— (2)

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

$= (n-2) * T(n-3)$ — (3)

Substitute (2) in (1)

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

Substitute (3) in (1)

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

$= n * (n-1) * (n-2) * T(n-k)$

$= n * (n-1) * (n-2) * T(1)$ where $n = k$

$= n * (n-1) * (n-2) * \ldots * 3 * 2 * 1$

$= n * n\left(1-\frac{1}{n}\right) * n\left(1-\frac{2}{n}\right) * \ldots n\left(\frac{3}{n}\right) * n\left(\frac{2}{n}\right) * n\left(\frac{1}{n}\right)$

$= n^n$

$= O(n^n)$

(14) Binary Search

$T(n) = T(n/2) + k$ if $n > 1$; 1 if $n = 1$

$$T(N) = T(N/2) + K \quad -(1)$$

$$T(N/2) = T\left(\frac{N/2}{2}\right) + K$$

$$= T(N/2^2) + K \quad -(2)$$

Substituting (2) in (1).

$$T(N) = \left(T\left(\frac{n}{2^2}\right) + k\right) + k.$$

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

Sub $n = n/2^2$ in (1).

$$T\left(n/2^2\right) = T\left(\frac{n/2^2}{2}\right) + K$$

$$= T\left(n/2^3\right) + k \quad -(4)$$

Sub (4) in (3)

$$T(N) = \left(T\left(n/2^3\right) + k\right) + 2k.$$

$$= T\left(n/2^3\right) + 3k \quad -(5)$$

$$T(N) = T\left(n/2^i\right) * i * k.$$

Let $n/2^i = 1$

$n = 2^i$

$\log(n) = \log(2^i)$

$\log(n) = i * \log 2$

$\log(n) = i$ (taking log base as 2)

$T(n) = 1 + \log(n) * K$

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

(5) $T(n) = 2T\left(\frac{n}{2}\right) + n$ — ① $T(1) = 1$

Replace $n \to \frac{n}{2}$ in eq ①

$T(n/2) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$ — ②

Substituting eq ② in ①

$T(n) = 2\left(2T(n/4) + n/2\right) + n$

$= 4T(n/4) + n + n$

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

Replace $n \to \frac{n}{4}$ in eq ① becomes

$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$ — ④

Substituting equation ④ in ③ we get,

$T(n) = 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n$

$= 8T\left(\frac{n}{8}\right) + n + 2n$

$= 2^3 T(n/2^3) + 3n$ — ⑤

$T(n) = 2^i T\left(\frac{n}{2^i}\right) + in$

Let $\frac{n}{2^i} = 1 \Rightarrow n = 2^i$

$\boxed{\log_2^n = i}$

general equation becomes

$$T(n) = n\,T(1) + n\log_2 n$$

$$T(n) = n + n\log_2 n$$

Time complexity $O(n \log n)$

(b)

Q: $T(n) = T(n-1) + n$ — (1) $T(1) = 1$

Replace $n \to n-1$

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

$= T(n-2) + (n-1)$ — (2)

Sub (2) in (1)

$T(n) = (T(n-2) + n-1) + n$ — (3)

Replace $n \to n-2$ in eq (1)

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

$= T(n-3) + n-2$ — (4)

Sub (4) in (3)

$T(n) = T(n-3) + (n-2) + (n-1) + n$ — (5)

$= T(1) + 2 + 3 + 4 + \ldots + n$

$= 1 + 2 + 3 + 4 + \ldots + n$

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

$O(n^2)$