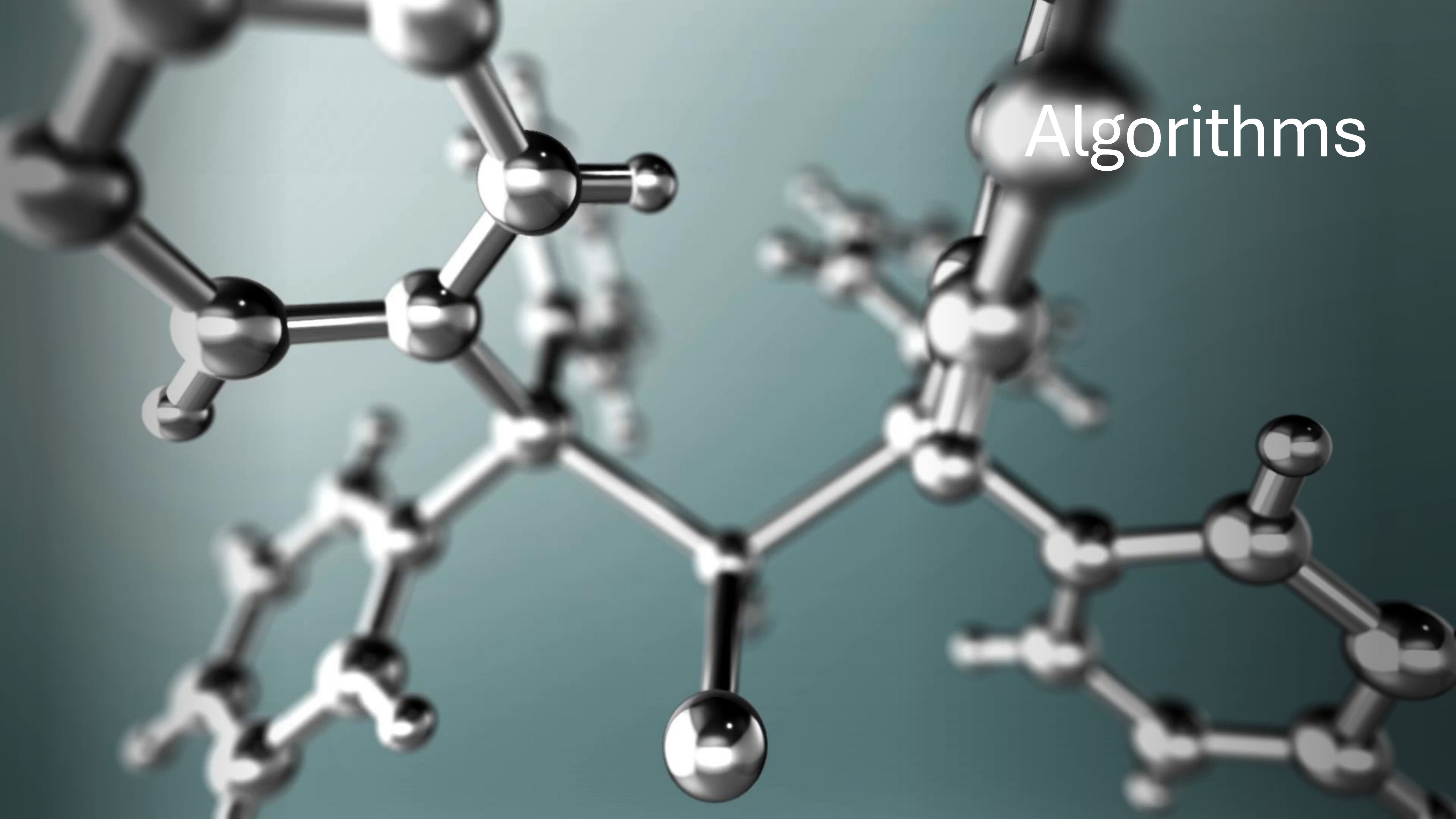
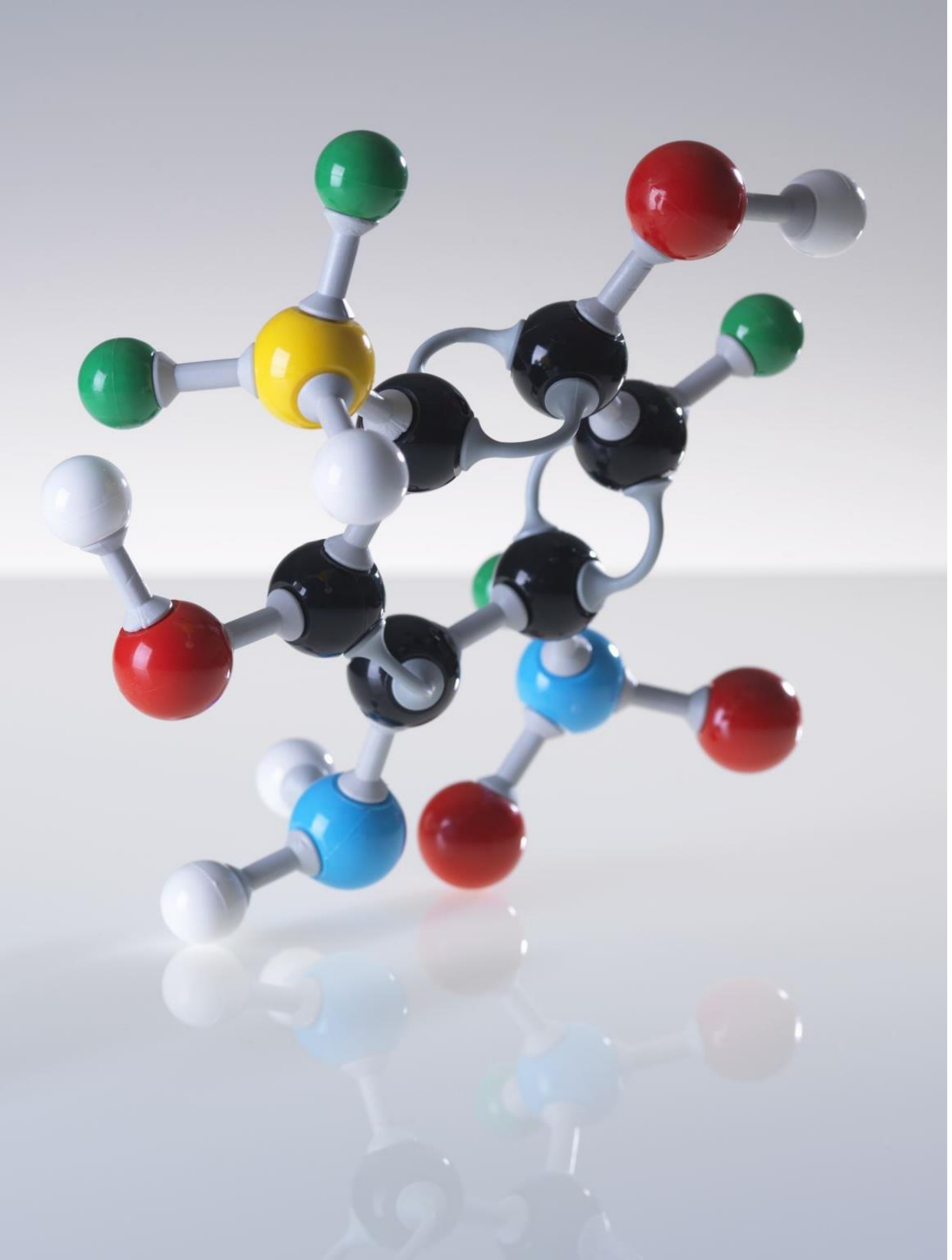
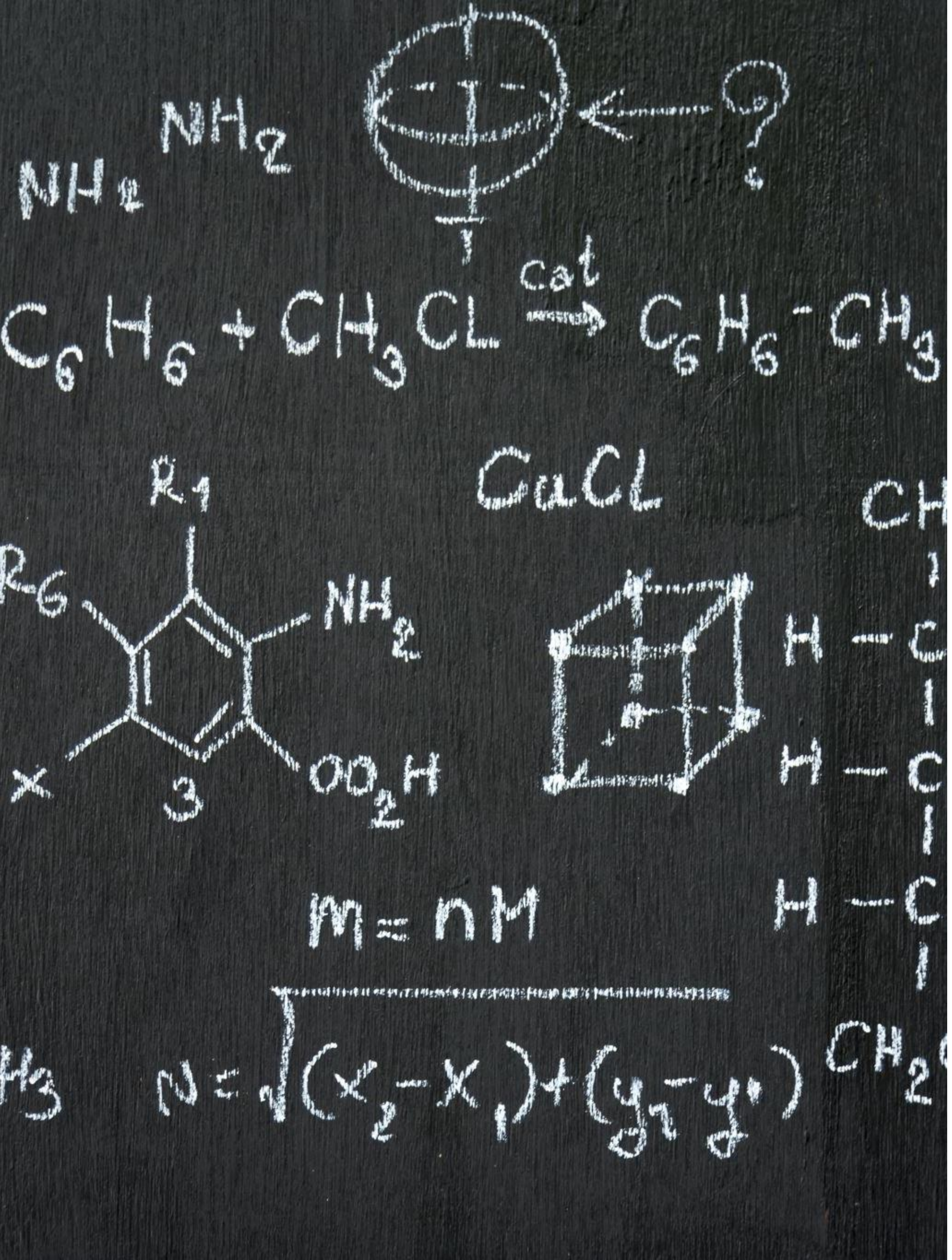


Algorithms



Algorithms

- An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.
- Algorithms can be described using English language, programming language, pseudocode



Algorithms (example)

- Describe an algorithm for finding the maximum value in a finite sequence of integers.

- Solution
- Set the temporary maximum equal to the first integer in the sequence.
- Compare the next integer in the sequence to the temporary maximum, and if it is larger than the temporary maximum, set the temporary maximum equal to this integer.
- Repeat the previous step if there are more integers in the sequence
- Stop when there are no integers left in the sequence.
- The temporary maximum at this point is the largest integer in the sequence.

Algorithms (example)

- Describe an algorithm for finding the maximum value in a finite sequence of integers. Solution:
- Procedure max(a1, a2, a3, ..., an: integers)
- max = a1 for i=2 to n if max < ai
    - then max = ai
    - output max
- Number of steps:
    - 1 + (n - 1) + (n - 1) + 1 = 2n

Growth of Functions

THE TIME REQUIRED TO SOLVE A PROBLEM DEPENDS ON THE NUMBER OF STEPS

G**ROWTH FUNCTIONS** ARE USED TO ESTIMATE THE NUMBER OF STEPS AN ALGORITHM USES AS ITS INPUT GROWS.

The largest number of steps needed to solve the given problem using an algorithm on input of specified size is **worst-case complexity**.

Example:

Design an algorithm to determine if finite sequence $a_1,a_2,\ldots,a_n$ has term 5.

```
Procedure search(a1, a2, a3, ..., an: integers) for i=1 to n if ai=5 then output True
output False
```

Worst-case complexity: $n + n + 1 = 2n + 1$

Worst-case complexity

Algorithm Complexity

To describe running time of algorithms, we consider the size of input is very large.

We can ignore constants in describing running time.

Linear Search

```
Procedure linear search(x: integer, a1, a2, a3, ..., an: integers)
While ( in and xai ) i = i + 1
if in then location = i
else location = 0
output location
Worst-case complexity:
2n + 2
```

Asymptotic notation

Allows for describing the behavior of algorithms and functions as their input size approaches infinity.

Provides a simple way to analyze and compare the efficiency of **algorithms** focusing on growth rates without any concerns with constant factors and lower-order terms.

Properties of Asymptotic Notation

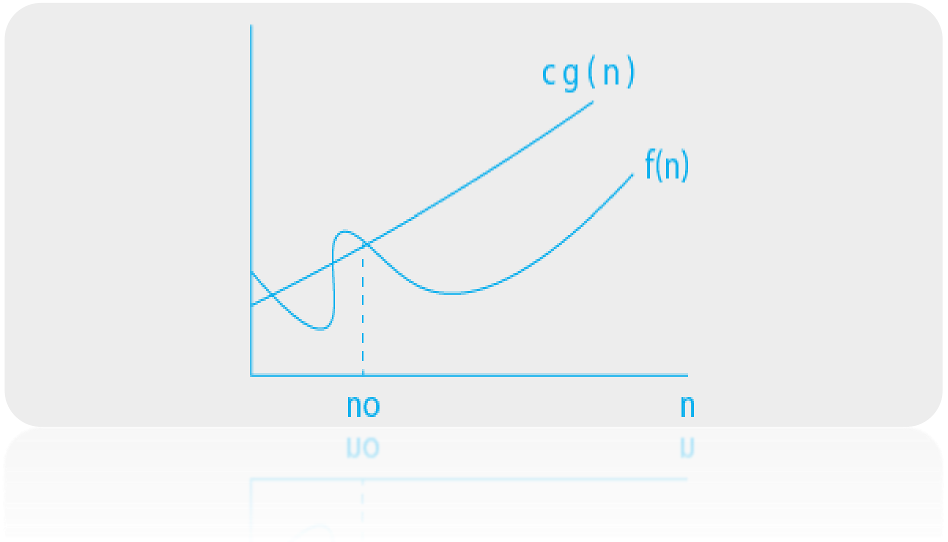
- BIG O
- Omega
- Theta

Knowing these properties will help in understanding the execution insights for:

- Best case
- Average Case
- Worst case

Big Oh Notation

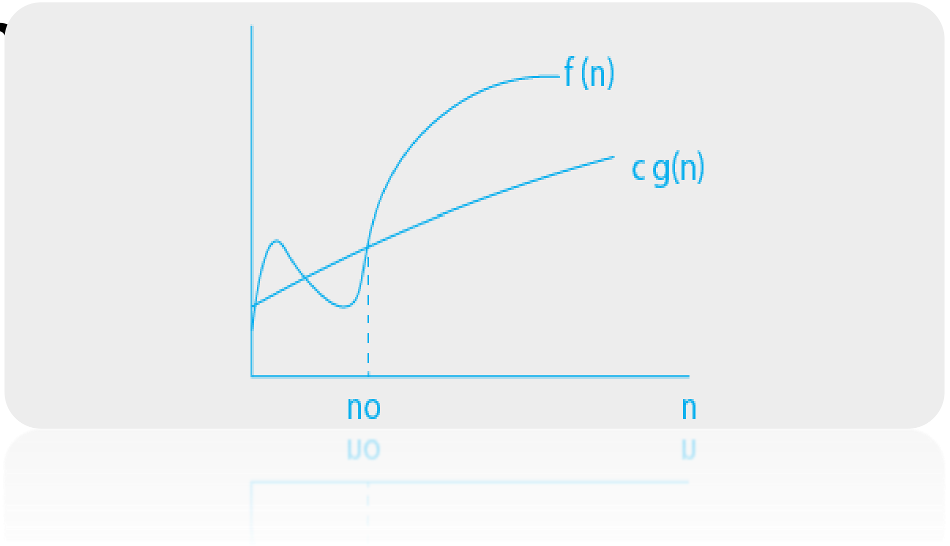
- The Asymptotic Notation O(n) represents the upper bound of an algorithm's running time.
- It measures or calculates the worst-case time complexity or the maximum or longest amount of time an algorithm can possibly take to complete.
- For example, O(log n) represents a binary search algorithm's upper bound.



g(n) is an Asymptotic upper bound for f(n)
O(g(n))={f(n): there exist a positive constants c and No such that 0<=f(n) <=cg(n)for all n>=No

Omega Asymptotic Notation

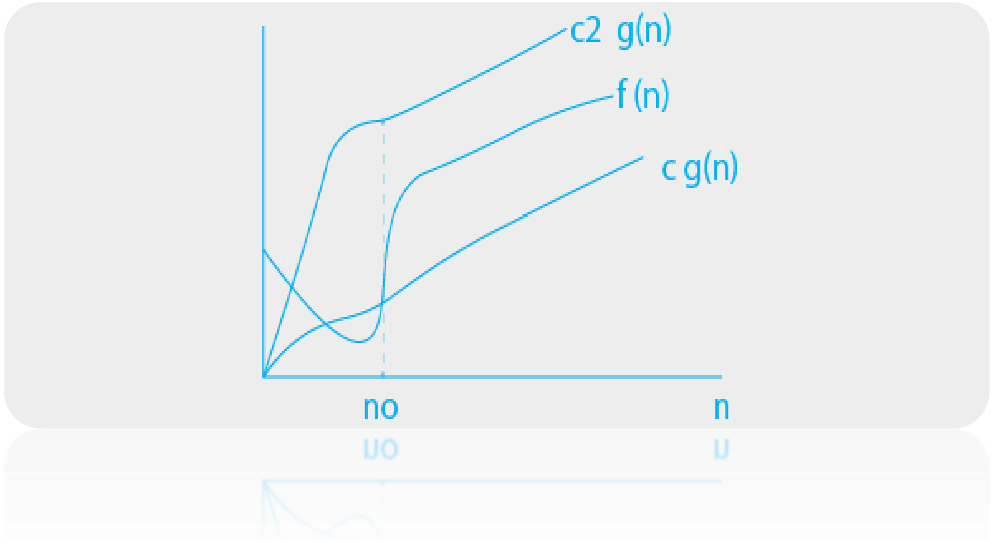
- The Asymptotic Notation O(n) represents the upper bound of an algorithm’s running time.
- It measures or calculates the worst-case time complexity or the maximum or longest amount of time an algorithm can possibly take to complete.
- For example, O(log n) represents a binary search algorithm’s upper bound.



g(n) is an asymptotic lower bound for f(n)
Ω(f(n)) ≥ { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n > n0. }

Theta Asymptotic Notation

- The Theta Asymptotic Notation Ο(n) represents the both lower bound and upper bound of an algorithm's running time. It measures the average case of time complexity. When we use big-Θ notation, we're saying that we have an asymptotically tight bound on the running time.



g(n) is an asymptotic tight bound for f(n)

The function f is said to be Θ(g), if there are constants c1, c2 > 0 and a natural number n0 such that c1 g(n) ≤ f(n) ≤ c2 g(n) for all n ≥ n0

Properties of Asymptotic Notation

- General Properties:
- If f(n) is O(g(n)) and k is constant then k*f(n) is also O(g(n)).

- Transitive Properties:
- If g(n) is O(h(n)) and f(n) is O(g(n)) then f(n) = O(h(n)).

- Reflexive Properties:
- If f(n) is given then f(n) is O(f(n)). Since the max value of f(n) will be f(n) itself
- Hence x = f(n) and y = O(f(n) tie themselves in reflexive relation always.

- Symmetric Properties:
- If f(n) is Θ(g(n)) then g(n) is Θ(f(n)).

- Transpose Symmetric Properties:
- If f(n) is O(g(n)) then g(n) is Ω (f(n)).

- Some More Properties:

- If f(n) = O(g(n)) and f(n) = Ω(g(n)) then f(n) = Θ(g(n))
- If f(n) = O(g(n)) and d(n)=O(e(n)) then f(n) + d(n) = O(max(g(n), e(n)))

Standard Notations and common functions

- Monotonicity: A function $f(n)$ is monotonically increasing if $m \leq n$ implies $f(m) \leq f(n)$.
- Similarly, it is monotonically decreasing if $m \leq n$ implies $f(m) \geq f(n)$.
- A function $f(n)$ is strictly increasing if $m < n$ implies $f(m) < f(n)$ and strictly decreasing if $m < n$ implies $f(m) > f(n)$.

Floors and Ceilings

- The biggest integer less than or equal to any real number x is denoted by bxc (also known as "the floor of x"), while the smallest integer larger than or equal to x is denoted by dxe (also known as "the ceiling of x").

Example- Linear Search

- Search for x in a given set of values
- Y1 y2 y3 yn

•**Procedure** linear search(x: integer, a_1, a_2, a_3, …, a_n: integers)

- **While** (in **and** xa_i) i = i + 1

•**if** in **then** location = i **else** location = 0 **output** location

•**Worst-case complexity:**

•**2n + 2**

Example- Binary Search

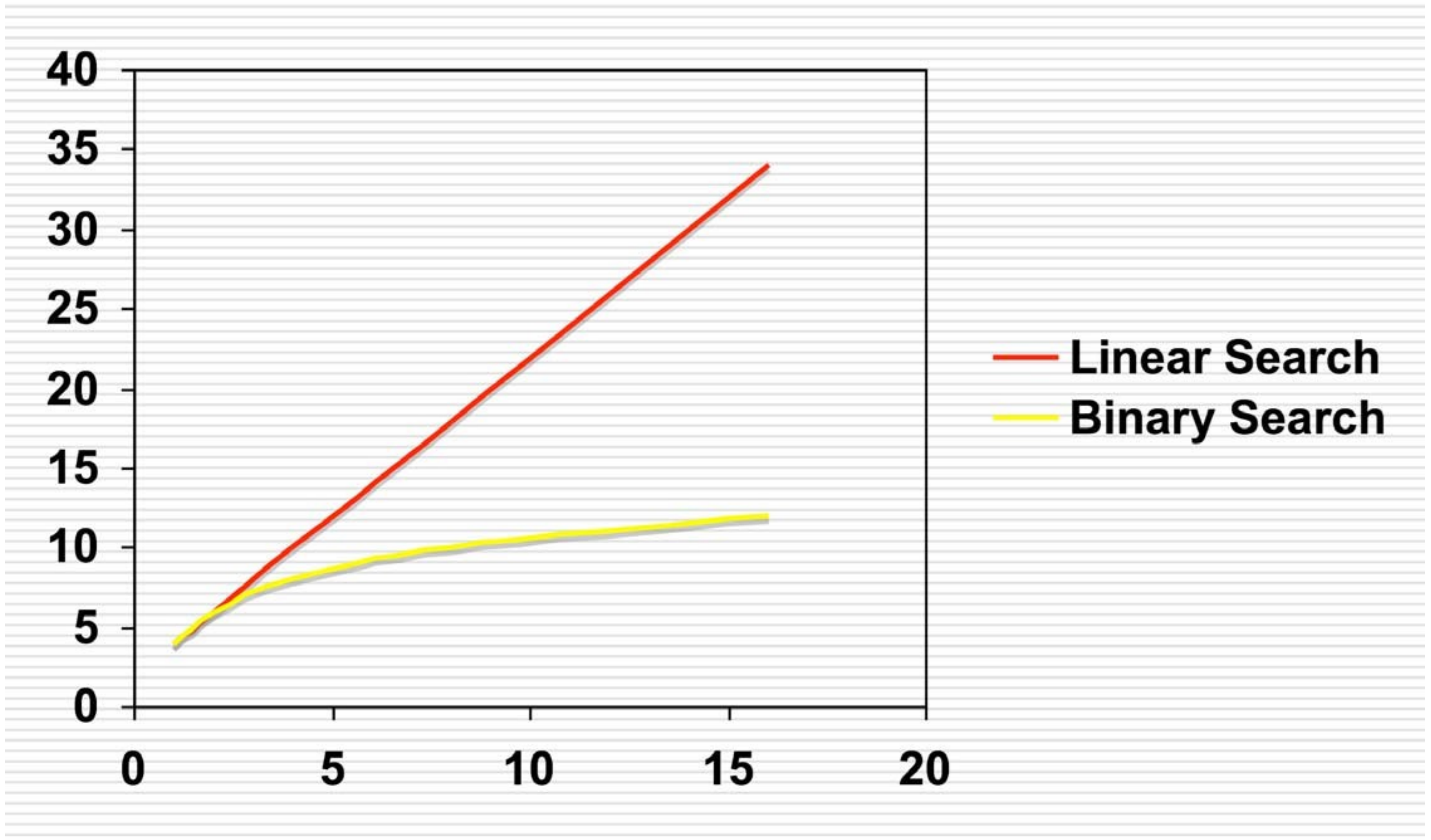
- Assume y_1 y_2, y_2 y_3, …, y_{n-1} y_n.
- Y1   y2   y3   yn
- Search for 15 in
    - $\overrightarrow{12 < 15\ ?}$
    - 1 3 5 12 15 19 20

      15=15
    - 1 3 5 12 15 19 20

```
Procedure binary search(x: integer, a1, a2, a3, ..., an: increasing integers)
i = 1 j = n m = (i+j)/2
while (amx and i  j) begin
          m = (i+j)/2
          if x > am then i = m+1 else j = m-1
          end
if x = am then location = m else location = 0 output location
```

Worst-case complexity:
3 + 3log(n) + 2

Linear Search vs. Binary Search



Big-O notation

Show that $f(x)=x^2 + 2x + 1$ is $O(x^2)$.

Solution:

$|f(x)| \ \ C|x^2|$ $x>k$

$|x^2 + 2x + 1| \ \ C(x^2)$ $x>k$

$x^2 + 2x + 1 \ \ x^2 + 2x^2 + x^2$ $x>1$

$x^2 + 2x + 1 \ \ 4(x^2)$ $x>1$

$k = 1$ and $C = 4$

$f(x) = O(x2)$ or $f(x) \ \ O(x2)$

$x^2 + 2x + 1 \ \ 4(x^2)$ $x^2 + 2x + 1 = O(x^2)$ $x^2 = O(x^2 + 2x + 1)$ $x>1$

$x^2 \ \ x^2 + 2x + 1$ $x>1$

So, x^2 and $x^2 + 2x + 1$ are of the same order.

Assume $f(x) = O(g(x))$ and $g(x) < g'(x)$ for large x. Show $f(x) = O(g'(x))$.

Proof:

$|f(x)| \ \ C|g(x)|$ $x>k$ $|f(x)| \ \ C|g(x)| < C|g'(x)|$ $x>k$ So, $f(x) = O(g'(x))$.

Big-O notation - Example

Show that $f(x)=7x^2$ is $O(x^3)$.

$|f(x)| \quad C|x^3|$ $\qquad\qquad\qquad\qquad$ $x>k$

7x2 7x3   x>1 k = 1 and C = 7

f(x) = O(x3) $\qquad\qquad$ or $\qquad\qquad$ f(x) O(x3)

Algorithm to swap two numbers

- 1. Start
- 2. Read value of A
- 3. Read value of B
- 4. C=A
- 5. A=B
- 6. B=C
- 7. Print A 8. Print B 9. End

Computation of a set of numbers

- Input : input number n
- Step 1: Start
- Step 2: Read number n Step 3: Declare sum to 0 and i to 1
- Step 4: Repeat steps 5 to 7 until i <= n
- Step 5: update sum as sum = sum + i
- Step 6: increment i
- Step 7: Print sum
- Step 8: Stop
- Output: sum

Factorial Computation

```
• Step 1: Start
• Step 2: Declare Variable n, fact, i
• Step 3: Read number from User
• Step 4: Initialize Variable fact=1 and i=1
• Step 5: Repeat Until i<=number
•           5.1 fact=fact*i
•           5.2 i=i+1
• Step 6: Print fact
• Step 7: Stop
```

Fibonacci Series

- Start
- Declare variables i, a,b , show
- Initialize the variables, a=0, b=1, and show =0
- Enter the number of terms of Fibonacci series to be printed
- Print First two terms of series
- Use loop for the following steps
- -> show=a+b
- -> a=b
- -> b=show
- -> increase value of i each time by 1
- -> print the value of show
- End

Reverse Digits of a number

- Input: num
- (1) Initialize rev_num = 0
- (2) Loop while num > 0
-     (a) Multiply rev_num by 10 and add remainder of num
-         divide by 10 to rev_num
-             rev_num = rev_num*10 + num%10;
-     (b) Divide num by 10
- (3) Return rev_num

Base Conversion Any base to Base 10

- Let n be the number of digits in the number. For example, 104 has 3 digits, so n=3.
- Let b be the base of the number. For example, 104 is decimal so b = 10.
- Let s be a running total, initially 0.
- For each digit in the number, working left to right do:
-   Subtract 1 from n.
-   Multiply the digit times bn and add it to s.
- When your done with all the digits in the number, its decimal value will be s

Binary to Base 10 – Ex: Binary Number 1011

- Let n = 4.
- Let b = 2.
- Let s = 0.
- First digit, 1: n = 3, 1 × bn is 1 × 23 = 8. So s = 8.
- Second digit, 0: n = 2, 0 × bn is 0 × 22 = 0. So s = 8.
- Third digit, 1: n = 1, 1 × bn is 1 × 21 = 2. So s = 10
- Last digit, 1: n = 0, 1 × bn is 1 × 20 = 1. So 10112 = 1110

Digit	n	Value = Digit * bn	Running Total
1	3	1 × 23 = 8	8
0	2	0 × 22 = 0	8
1	1	1 × 21 = 2	10
1	0	1 × 20 = 1	11

Hex to Base 10 7E

- Let n = 2.
- Let b = 16.
- Let s = 0.
- First digit, 7: n = 1, 7 × bn is 7 × 161 = 7 × 16 = 112. So s = 112.
- Last digit, E: n = 0, 14 × bn is 14 × 160 = 14. So s = 112 + 14 = 126. So 7E16 = 12610

Digit	n	Value = Digit * bn	Running Total
7	1	7 × 161 = 112	114
E	0	14 × 160 = 14	126

Octal Number 124

- Let n = 3.
- Let b = 8.
- Let s = 0.
- First digit, 1: n = 2, 1 × bn is 1 × 82 = 1 × 64 = 64. So s = 64.
- Second digit, 2: n = 1, 2 × bn is 2 × 81 = 2 × 8 = 16. So s = 64 + 16 = 80.
- Last digit, 4: n = 0, 4 × bn is 4 × 80 = 4. So s = 80 + 4 = 84. So 1248 = 8410

Digit	n	Value = Digit * bn	Running Total
1	2	1 × 82 = 64	64
2	1	2 × 81 = 16	80
4	0	4 × 80 = 4	84

From Decimal To Another Base

- Let n be the decimal number.
- Let m be the number, initially empty, that we are converting to. We'll be composing it right to left.
- Let b be the base of the number we are converting to.
- Repeat until n becomes 0
- Divide n by b, letting the result be d and the remainder be r.
- Write the remainder, r, as the leftmost digit of b.
- Let d be the new value of n.

45 into Binary

- Let n = 45.
- Let b = 2.
- Repeat
- 45 divided by b is 45/2 = 22 remainder 1. So d=22 and r=1. So m= 1 and the new n is 22.
- 22 divided by b is 22/2 = 11 remainder 0. So d=11 and r=1. So m= 01 and the new n is 11.
- 11 divided by b is 11/2 = 5 remainder 1. So d=5 and r=1. So m= 101 and the new n is 5.
- 5 divided by b is 5/2 = 2 remainder 1. So d=2 and r=1. So m= 1101 and the new n is 2.
- 2 divided by b is 2/2 = 1 remainder 0. So d=1 and r=0. So m= 01101 and the new n is 1.
- 1 divided by b is 1/2 = 0 remainder 1. So d=0 and r=1. So m=101101 and the new n is 0. So 4510 = 1011012

99 Into Binary

- Let n = 99.
- Let b = 2.
- Repeat
- 99 divided by b is 99/2 = 49 remainder 1. So d=49 and r=1. So m= 1 and the new n is 49.
- 49 divided by b is 49/2 = 24 remainder 1. So d=24 and r=1. So m= 11 and the new n is 24.
- 24 divided by b is 24/2 = 12 remainder 0. So d=12 and r=0. So m= 011 and the new n is 12.
- 12 divided by b is 12/2 = 6 remainder 0. So d=6 and r=0. So m= 0011 and the new n is 6.
- 6 divided by b is 6/2 = 3 remainder 0. So d=3 and r=0. So m= 00011 and the new n is 3.
- 3 divided by b is 3/2 = 1 remainder 1. So d=1 and r=1. So m= 100011 and the new n is 1.
- 1 divided by b is 1/2 = 0 remainder 1. So d=0 and r=1. So m=1100011 and the new n is 0.
  So 9910 = 11000112

45 into HexaDecimal

- Let n = 45.
- Let b = 16.
- Repeat
- 45 divided by b is 45/16 = 2 remainder 13. So d=2 and r=13. So m= D and the new n is 2.
- 2 divided by b is 2/16 = 0 remainder 2. So d=0 and r=2. So m=2D and the new n is 0. So 4510 = 2D16.

Strings/Char to number and back again

- Characters are the things that we recognize as having some meaning.
- Since a computer only deals with 0/1 (binary numbers), there is a mapping between characters and character-codes (number).
- The mapping commonly used in computers is ASCII. ASCII allocates 8 bits per character, but only uses 7, resulting 128 characters

Char/String to Numbers

- **Character-codes and digits**
- When handling set of characters that the human interprets as a number, one must convert the character to a digit.

- The digit is the number associated with the character.
- So 'C' in base 16 will become a digit 12 (xC). : ***Note that the character-code is NOT the digit value.***
- The character-code of 'C' is the bit pattern: 0100.0011 (x43).
- The digit C has the bit pattern: 0000.1100 (x0C). The conversion from character-code to digit is:
- digit = character-code -'0'; for '0' to '9'
- digit = character-code -'A' + 10; for 'A' to 'Z'
- This relies on the fact that in the ascii/unicode character-codes of '0' to '9' and 'A' to 'Z' have consecutive, increasing values.

Why Convert ?

- Consider the string "2573" and the number 2573. Both can be stored in 32 bits.
- The string is four 8 bit ascii characters, and an int is typically 32 bits on a computer.
- The internal format is different.
- 0011-0010|0011-0101|0011-0111|0011-0101 (the four ascii chars "2573")
- 0000-0000-0000-0000-0000-1010-0000-1101 (the 2's complement value 2573)

Example

- Converting string to numbers can be done one character at a time. Process the characters from left to right and build up the final value.
- The basic steps are:
- initialize: value = 0
- loop over the characters in the string, left to right
- convert the character to a digit
- compute: value = value * base + digit
- repeat steps 3 and 4 for all the characters
- value now contains the correct number
- When the base is a power of 2, the multiplication by base can be implemented by a simple left shift operation (e.g. shift by 4 for hex, 3 for octal, etc).

How does this work – Honer’s rule for polynomial evaluation

- Given a polynomial of the form $c_nx^n + c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + \ldots + c1x + c0$ and a value of x, find the value of polynomial for a given value of x. Here c_n, c_{n-1}, .. are integers (may be negative) and n is a positive integer.
- Input is in the form of an array say poly[] where poly[0] represents coefficient for xn and poly[1] represents coefficient for xn-1 and so on.

Examples

- // Evaluate value of 2x3 - 6x2 + 2x - 1 for x = 3
- Input: poly[] = {2, -6, 2, -1}, x = 3
- Output: 5

- // Evaluate value of 2x3 + 3x + 1 for x = 2
- Input: poly[] = {2, 0, 3, 1}, x = 2
- Output: 23

Example

- Convert the string "263" to decimal.
- value = 0
- "263" - value = 10 * 0 + 2 = 2
- "263" - value = 10 * 2 + 6 = 26
- "263" - value = 10 * 26 + 3 = 263

Binary to String

- Convert the string "010011" to its decimal value.value = 0
- "**0**10011" - value = 2 * 0 + 0 = 0
- "0**1**0011" - value = 2 * 0 + 1 = 1
- "01**0**011" - value = 2 * 1 + 0 = 2
- "010**0**11" - value = 2 * 2 + 0 = 4
- "0100**1**1" - value = 2 * 4 + 1 = 9
- "01001**1**" - value = 2 * 9 + 1 = 19

String to hex

- Convert the string "C8A" to decimal.value = 0
- "**C**8A" **-** value = 16 * 0 + 12 = 12
- "C**8**A" **-** value = 16 * 12 + 8 = 200
- "C8**A**" **-** value = 16 * 200 + 10 = 3210

Growth of Polynomial Functions

The leading term of a polynomial function determines its growth.

Let $f(x)=a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$, where $a_n, a_{n-1}, \ldots, a_1, a_0$ are real numbers.

Then f(x) is **$O(x^n)$.**

Let f(x)=anxn + an-1xn-1 + ... + a1x + a0, where an, an-1, ..., a1, a0 are real numbers.
Show f(x) is O(xn).

Proof:
Show (C,k) that |f(x)| C|g(x)| x>k.
|f(x)| = | anxn + an-1xn-1 + ... + a1x + a0|
Assume x>1.
|f(x)| = | an| xn + |an-1| xn-1 + ... + |a1| x + |a0|
 = xn (| an| + |an-1| /x + ... + |a1|/ xn-1 + |a0|/ xn)
  xn (| an| + |an-1| + ... + |a1| + |a0|) Let C = | an| + |an-1| + ... + |a1| + |a0| and k=1.
So, |f(x)| C|g(x)| x>k and f(x)=O(xn).

Give Big-O estimates for the factorial function f(n)=n! and g(x) = log(n!)

(n! = 1 . 2 . 3 . … . (n-1) . n)

Proof:

n! = 1 . 2 . 3 . … . (n-1) . n   n . n . n . … . n . n = n^n f(n) = O(n^n) taking C=1 and k=1.

log n! log n^n = n log n g(n) = O(n log n) taking C=1 and k=1.

Show n = O(2^n) and log n = O(n).

Proof:

n < 2^n where k=1 and C=1 n = O(2n)

n < 2n log n < log 2^n = n log 2 log n < n log 2 where k=1 and C=log 2 log n = O(n)

Show $\log_b$ n = O(n).

Proof:

$\log_b$ n = log n / log b < n / log b

$\log_b$ n < n / log b where k=1 and C=1 / log b $\log_b$ n = O(n)

Big-O notation - Example

Is it true that x3 is O(7x2).

Solution:

Determine whether witnesses exist or not.

x3 C(7x2) whenever x>k x 7C whenever x>k

No matter what C and k are, the inequality x 7C cannot hold for all n with n>k.

So, x3 is not O(7x2).

Notes and Summary

- **What is asymptotic notation?**
  Asymptotic notation is a mathematical framework used to analyze the performance of algorithms and functions as their input size approaches infinity.
- It abstracts away constant factors and lower-order terms, focusing on the fundamental growth rates of these algorithms and functions.

Notes and Summary

- **What is Big O notation?**
  Big O notation, often denoted as O(), describes the upper bound or worst-case behavior of an algorithm. It provides an estimation of how an algorithm’s runtime grows in relation to the input size, ignoring constant factors and lower-order terms.

- **What is Omega notation?**
  Omega notation, represented as Ω(), signifies the lower bound or best-case behavior of an algorithm. It indicates the minimum rate at which the algorithm’s runtime will grow with increasing input size.

- **What is Theta notation?**
  Theta notation, denoted as Θ(), combines both upper and lower bounds to provide a tight range of growth rates for an algorithm.
- It gives a more precise description of an algorithm’s performance than Big O or Omega notation alone.

Notes and Summary

- **Why are asymptotic notations important?**
- Asymptotic notations provide a standardized way to compare and analyze algorithms' efficiency, making it easier to assess their scalability and performance characteristics.

- They help us focus on the core trends in algorithm behavior as input sizes become large, which is essential for designing robust and efficient software.