

19CSE111 Fundamentals of Data Structures

Unit I

Course Objectives

- This course aims to
 - give the students an introduction to the structure and functionalities of the common data structures used in computer science
 - solve simple problems by applying the properties and functionalities of these data structures

Course Outcomes

- CO1: Understand the basics of analysis of algorithms
- CO2: Understand the linear data structures, hash tables and their functionalities
- CO3: Solve simple problems that uses the properties and functions of the data structures

Unit 1 - Syllabus

Basic concepts of Data Structures; Basic Analysis of Algorithms - big-Oh notation, efficiency of algorithms, notion of time and space complexity. Stacks: properties, LIFO, functions, Simple problems.

What is Data?

- Data are collections of binary elements processed and transmitted electronically by technologies like computers
- Data can be
 - Qualitative: descriptive information
 - Quantitative: numerical information

Quantitative Vs. Qualitative

Example: What do we know about Arrow the Dog?

Qualitative:

- He is brown and black
- He has long hair
- He has lots of energy

Quantitative:

- Discrete:
 - He has 4 legs
 - He has 2 brothers
- Continuous:
 - He weighs 25.5 kg
 - He is 565 mm tall



“Discrete data is counted, Continuous data is measured”

Information

- Data is a raw and unorganized fact that is required to be processed to make it meaningful
- Information is a set of data which is processed in a meaningful way according to the given requirement
 - Information is processed, structured, or presented in a given context to make it meaningful and useful

DIKW (Data – Information – Knowledge - Wisdom)

Data

- 100

Information

- 100 Kilometers

Knowledge

- 100 Km is quite a far distance

Wisdom

- Walking 100 Km is very difficult by any person, but vehicle transport is okay

Data Structures

- A data structure is a systematic way of organizing data in computer's memory
- Why is data structure necessary?
 - To help computer programs
 - Store data
 - Retrieve data
 - and Perform computations on data
 - in an effective manner

Data Structure's Definition

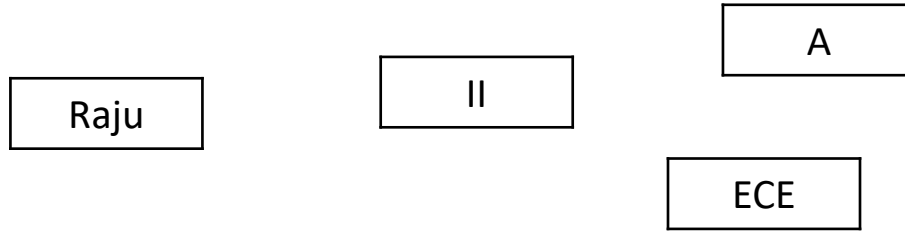
- [Wikipedia](#): A data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data
- [Britannica](#): A data structure is way in which data are stored for efficient search and retrieval
- [GeeksForGeeks](#): A data structure is a particular way of organizing data in a computer so that it can be used effectively
- [Studytonight](#): Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way

A close-up view of a white shelf filled with colorful binders and books. The binders are arranged vertically, with labels visible on their spines. A person's hands are visible at the bottom right, reaching towards the binders. The labels on the binders include "MANUALS", "GOALS & DREAMS", "REAL ESTATE NOTES", "STAFF", "DIVIDENDS", "CHRISTMAS", "FAVORITE RECIPES", "LEGAL & CONTRACTS", "LICENSES & INSURANCE", "ATTY HANDBOOK", "NANO CONFERENCE 2010", and "MANGET RETREAT". Above the binders, there is a row of books, some of which have titles like "book", "Business", "In No Time", "Analytics", "ed home", "Scott McLean-Parks", "Management", "AND WOMEN", "WOMEN", "WALL COOL SPACES", and "rehabment".

Pic. Credit: Pinterest.com

The need for Data Structures

- Rendering data elements in terms of some relationships for better organization
- Suppose we have some data

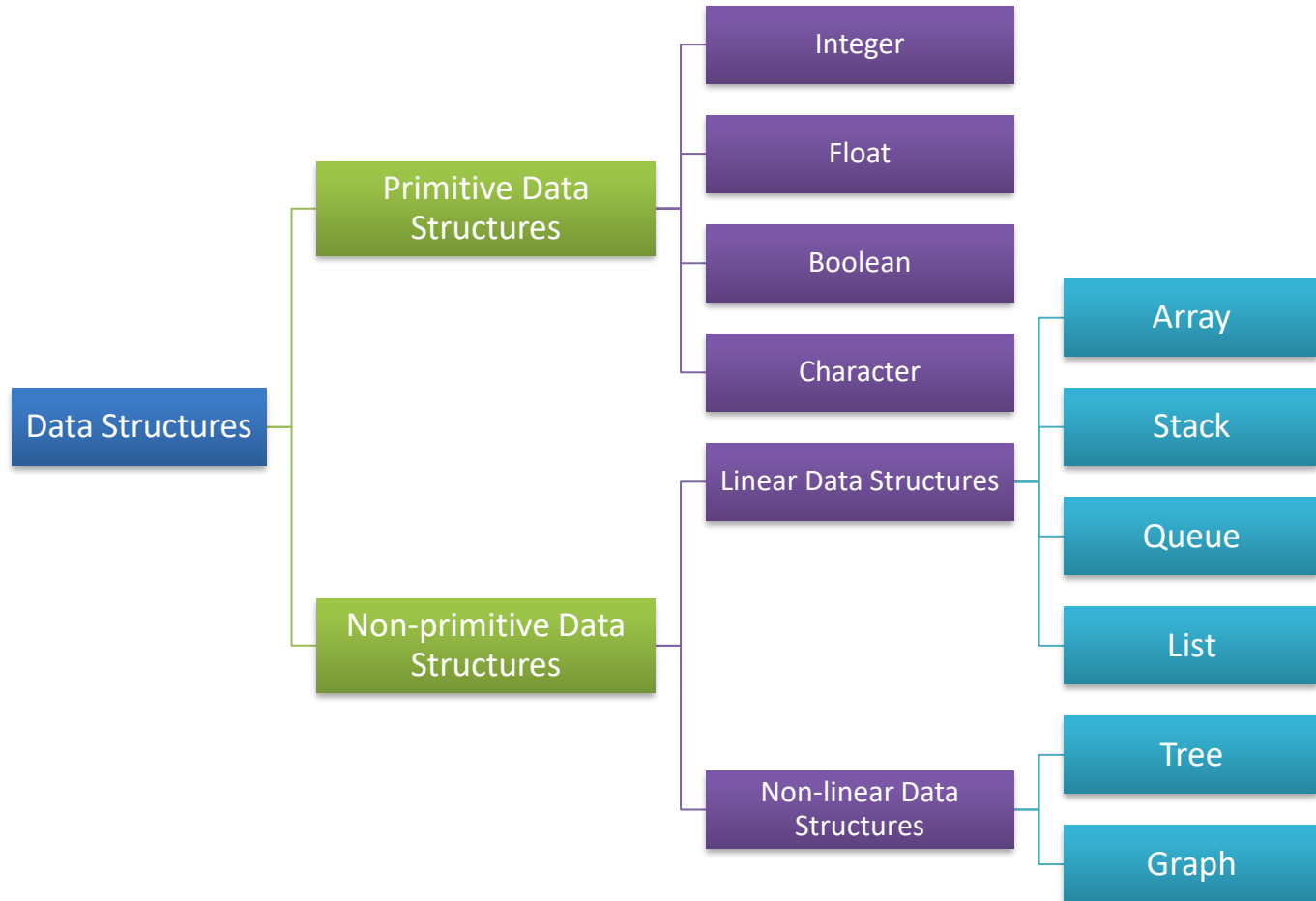


- This data can be organized like a Student record as below:

Name	Year	Sec	Branch
Raju	II	A	ECE

- Records like this can be collected and stored in a file or database

Data Structures Classification



Primitive Vs. Non-primitive Data Structure

- Primitive
 - Basic structures that are directly operated upon by machine instructions
 - Different representations on different computers
- Non-primitive
 - More sophisticated and derived from primitive data structures
 - They emphasize on grouping of data elements of same or different types

How do I classify Data Structures?

- Data Structures can be classified according to the following characteristics:

Characteristic	Description
Linear	Data arranged in a linear sequence e.g. array
Non-linear	Data arranged in a non-linear fashion e.g. tree
Homogeneous	Data elements of the same type e.g. array
Non-Homogeneous	Data elements of different types e.g. structure
Static	The size is fixed (static memory allocation) e.g. array
Dynamic	The size is variable (dynamic memory allocation) e.g. linked list

Operations on Data Structure

Operation	Description
Traversing	Visiting each data element of the data structure
Insertion	Adding an element to the data structure
Deletion	Removing an element from the data structure
Searching	Finding the location of an element within the data structure
Sorting	Arranging the elements in specific order
Selection	Selecting a specific element
Merging	Joining the data elements

Algorithms

- An algorithm is a step-by-step procedure for performing some task in a finite amount of time
- Features of a good algorithm
 - **Precision:** a good algorithm must have a certain steps that should be exact enough
 - **Uniqueness:** each step taken in the algorithm should give a definite result as stated
 - **Feasibility:** the algorithm should be possible and practicable in real life
 - **Input:** a good algorithm must be able to accept zero or more inputs
 - **Output:** a good algorithm must be able to produce results as output
 - **Finiteness:** the algorithm should have a stop after a certain number of instructions

Algorithm Analysis

-Eureka!



Interested in the
design of “good” data
structures and
algorithms

Algorithm Analysis

- Data Structures and Algorithms are central to computing
 - To classify them as good or bad, we need a precise of analyzing them
 - The primary analysis tool is

“Characterizing the running times of algorithms and data structure operations, with space usage also being of interest”

Algorithm Analysis

- **Running time**
 - A natural measure
 - Varying with input size and different inputs of the same size
 - Hardware and Software dependent
 - Also, translator dependent
- **Space**
 - Total space occupied by the algorithm
 - Includes auxiliary space

Experimental Analysis

- Experimental studies

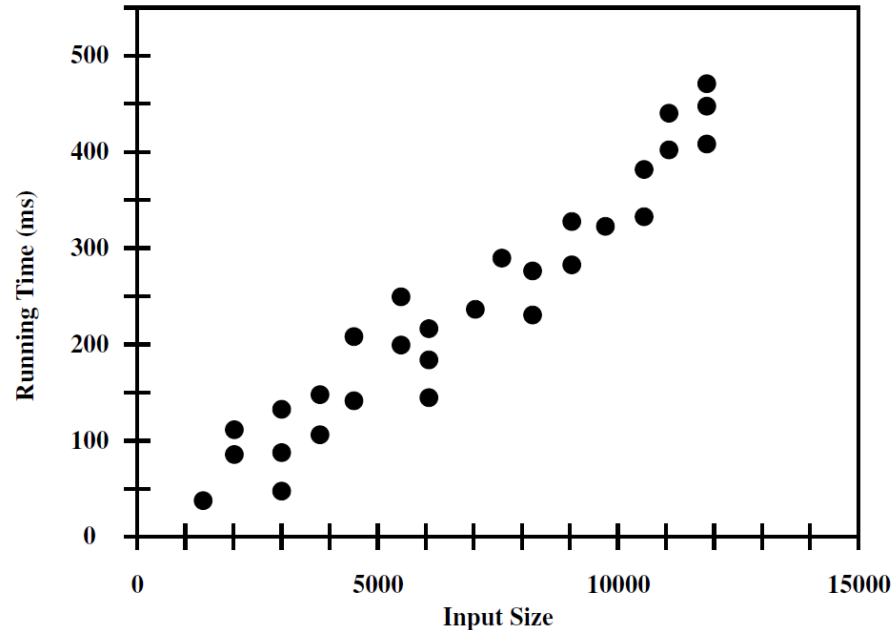
- Executing the algorithm on various inputs and recording the time

```
from time import time
start_time = time( )           # record the starting time
run algorithm
end_time = time( )             # record the ending time
elapsed = end_time - start_time # compute the elapsed time
```

- The time function measures relative to what is known as the “wall clock”
- The elapsed time will depend on other processes that running in parallel
- **Fairer Metric:** The number of CPU cycles that are used by the algorithm

Experimental Analysis

- The general dependence of running time on the size and structure of the input
 - Performing independent experiments on many different test inputs of various sizes and visualizing through plots



Challenges of Experimental Analysis

- Three major limitations of experimental studies' use to algorithm analysis:
 - Experimental running time is hardware and software dependent
 - Limitation on input set
 - Full implementation is required

Moving Beyond Experimental Analysis

- We need an approach to analyzing the efficiency of algorithms that:
 - Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment
 - Is performed by studying a high-level description of the algorithm without need for implementation
 - Takes into account all possible inputs

Counting Primitive Operations

- Performing analysis directly on a high-level description of the algorithm
- We define a set of primitive operations such as the following:
 - Assigning an identifier to an object
 - Determining the object associated with an identifier
 - Performing an arithmetic operation
 - Comparing two numbers
 - Accessing a single element of a Python list by index
 - Calling a function (excluding operations executed within the function)
 - Returning from a function

Primitive Operation

- A primitive operation corresponds to a low-level instruction with an execution time that is constant
 - Might be the type of basic operation executed by the computer hardware
 - We will simply count how many primitive operations are executed (t)
 - The operation count t will correlate to an actual running time in a specific computer
 - **Assumption:** The running times of different primitive operations will be fairly similar

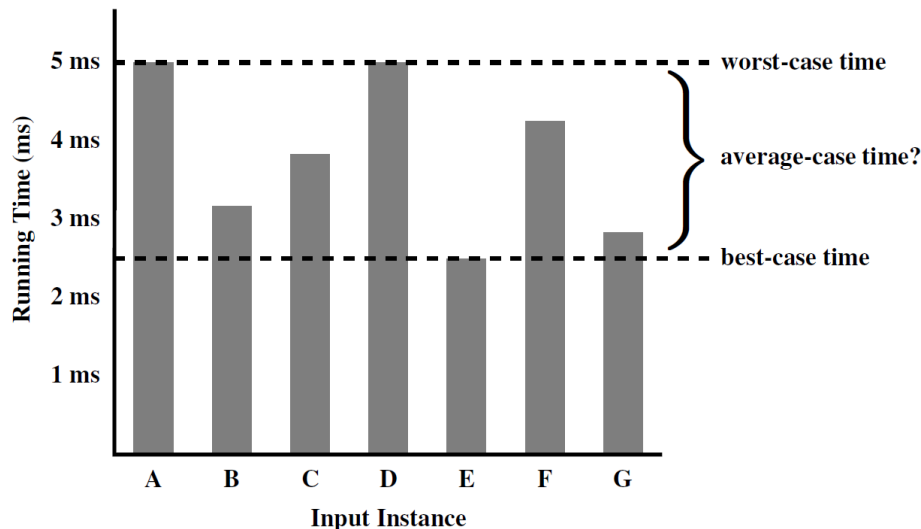
The number, t , of primitive operations an algorithm performs will be proportional to the actual running time of that algorithm

The Average-case Input

- An algorithm may run faster on some inputs than it does on others of the same size

$$t = f(n)$$

- n : the average over all possible inputs of the same size
- such an average-case analysis is typically quite challenging



The Worst-case Input

- The case that causes a maximum number of operations to be executed
 - It requires only the ability to identify the worst-case input
 - Designing for the worst case leads to stronger algorithmic “muscles”



Functions used in Algorithm Analysis - 1

- The Constant Function

- The simplest function we can think of

$$f(n) = c$$

- It does not matter what the value of n is; $f(n)$ will always be equal to the constant value c
- As simple as it is, the constant function is useful in algorithm analysis

Functions used in Algorithm Analysis - 2

- The Logarithm Function

- The logarithm function is ubiquitously present in algorithm analysis

$$f(n) = \log_b n, \text{ for some constant } b > 1$$

- This is define as

$$x = \log_b n \text{ if and only if } b^x = n$$

$$\begin{aligned} \log_2 8 &= 3 \\ 2^3 &= 8 \\ 2^{3.2} &= 9 \end{aligned}$$

- Logarithm Rule

- Given real numbers $a > 0$, $b > 1$, $c > 0$ and $d > 1$, we have

1. $\log_b(ac) = \log_b a + \log_b c$
2. $\log_b(a/c) = \log_b a - \log_b c$
3. $\log_b(a^c) = c \log_b a$
4. $\log_b a = \log_d a / \log_d b$
5. $b^{\log_d a} = a^{\log_d b}$

Logarithm Examples

- $\log(2n)$ ✓
- $\log(n/2)$ ✓
- $\log n^3$ ✓
- $\log 2^n$
- $\log_4 n$
- $2^{\log n}$ ✓

1

2

1
2
2

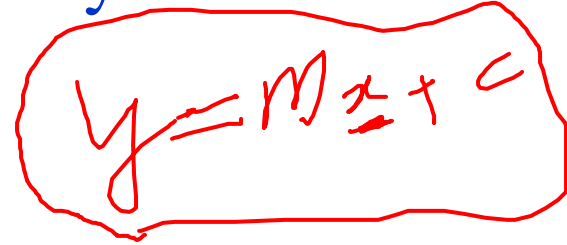
Functions used in Algorithm Analysis - 3

- The Linear Function

- Another simple yet important function

$$f(n) = n$$

- This function arises in algorithm analysis any time we have to do a single basic operation for each of n elements
- It represents the best running time we can hope to achieve



A handwritten red equation $y = mx + c$ is enclosed in a red oval. The 'x' has a small horizontal line through it, and the 'c' is a simple constant term.

Functions used in Algorithm Analysis - 4

- The N-Log-N Function

- The function that assigns to an input n the value of n times the logarithm base-two of n

$$n < f(n) = n \log_2 n < n^2$$

- Grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function
- An algorithm with a running time that is proportional to $n \log n$ is a great preference

Functions used in Algorithm Analysis - 5

- Quadratic Function

- Another function that appears often in algorithm analysis

$$f(n) = n^2$$

- Given an input value n , the function f assigns the product of n with itself
- It appears in algorithm analysis because there are many algorithms that have nested loops

Handwritten red text illustrating nested loops and their time complexity:

$$\begin{aligned} f(: i < n;) & \quad \text{--- } O - n \\ f(: j < n;) & \quad \text{--- } \frac{n^2}{2} \end{aligned}$$

The diagram shows two lines of code. The first line, $f(: i < n;)$, is followed by a horizontal line and then $O - n$. The second line, $f(: j < n;)$, is followed by a horizontal line and then $\frac{n^2}{2}$. A large red arrow points from the $\frac{n^2}{2}$ term towards the bottom right of the slide.

Functions used in Algorithm Analysis - 6

- The Cubic Function and Other Polynomials

- It assigns to an input value n the product of n with itself three times

$$f(n) = n^3$$

- It does appear from time to time
- The functions seen before are viewed as a part of a larger class of functions, the polynomials

$$f(n) = a_0 + a_1n + a_2n^2 + a_3n^3 + \dots + a_dn^d$$

- $a_0, a_1, a_2, \dots, a_d$: *coefficients*
- d : *the degree of the polynomial*

Handwritten notes:
 $i < n$
1 - $f(i)$
2 - $f(j)$ ($j < n$)
3 - $f(k)$ ($k < n$)

Running times that are polynomials with small degree are generally better than polynomial running times with larger degree.

Functions used in Algorithm Analysis - 7

- The Exponential Function

- It assigns to the input argument n the value obtained by multiplying the base b by itself n times

$$f(n) = b^n$$

- where b is a positive constant, called the base, and the argument n is the exponent
- 2 is the most common base
- Exponent Rules:

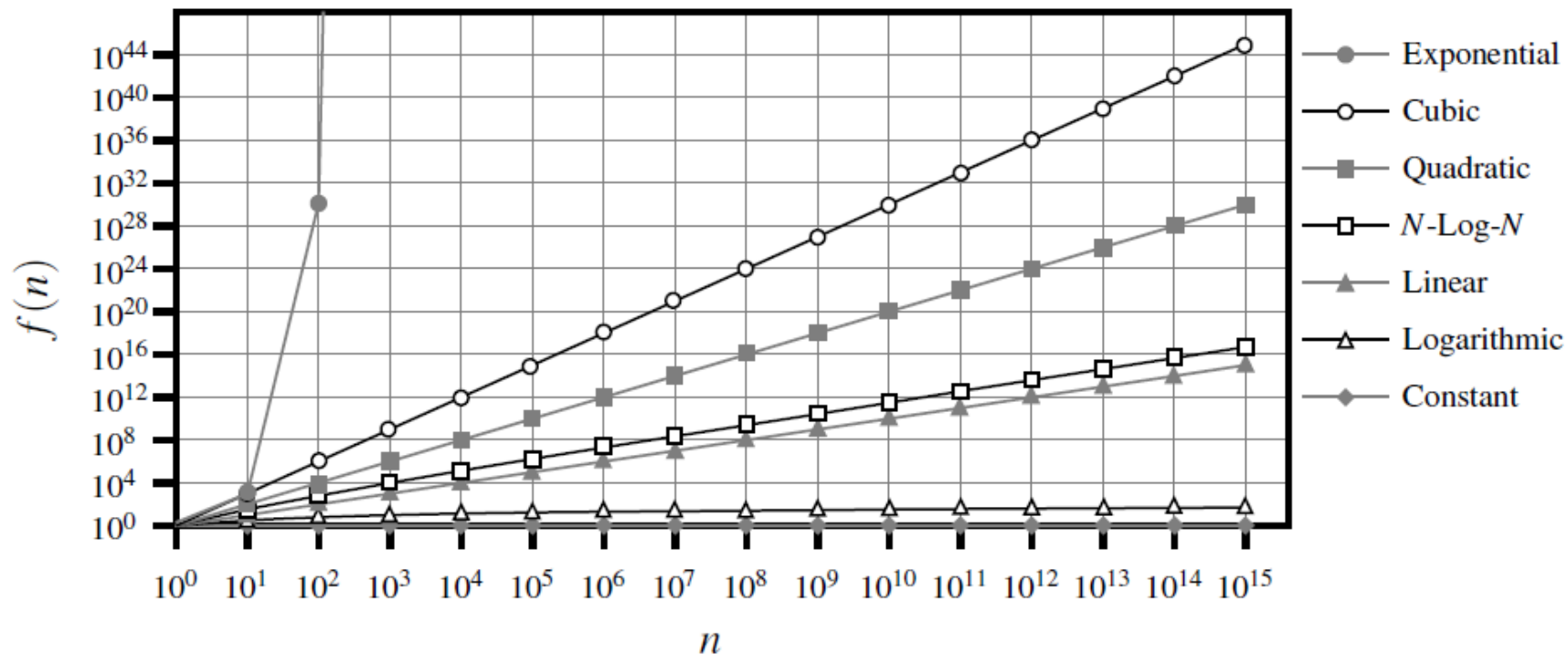
1. $(b^a)^c = b^{ac}$

2. $b^a b^c = b^{a+c}$

3. $b^a / b^c = b^{a-c}$

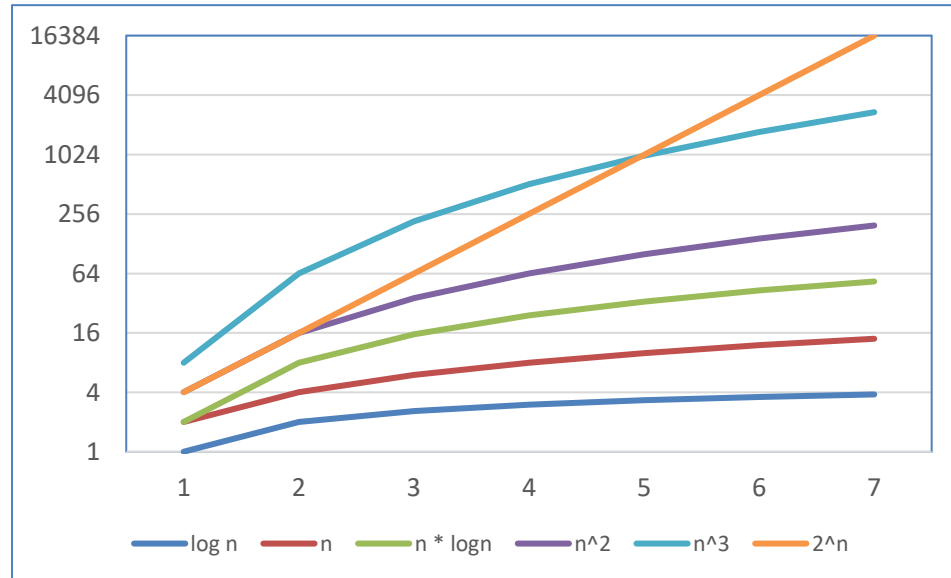
Comparing Growth Rates

constant	logarithm	linear	n -log- n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n



Comparative Analysis

Input Size	$\log n$	n	$n * \log n$	n^2	n^3	2^n
2	1	2	2	4	8	4
4	2	4	8	16	64	16
6	2.584962501	6	15.509775	36	216	64
8	3	8	24	64	512	256
10	3.321928095	10	33.21928095	100	1000	1024
12	3.584962501	12	43.01955001	144	1728	4096
14	3.807354922	14	53.30296891	196	2744	16384



Asymptotic Analysis

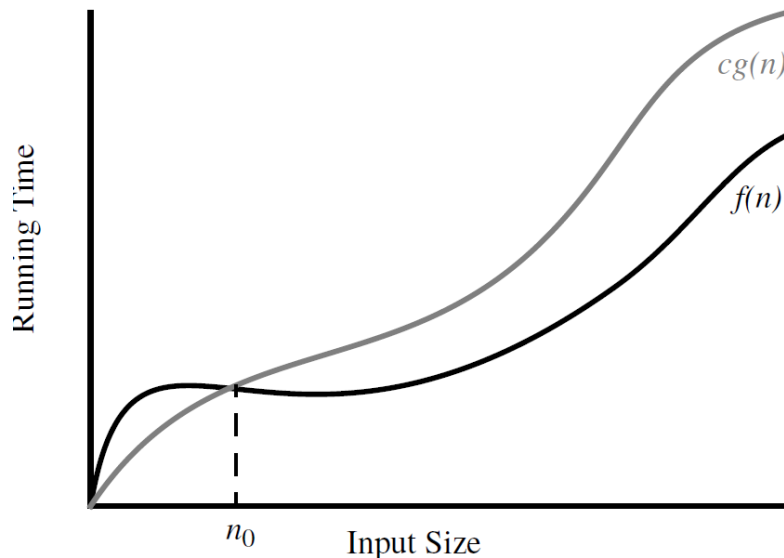
- A “big-picture” approach
 - the running time of an algorithm grows proportionally to n
 - each basic step in a pseudo-code description or a high-level language implementation may correspond to a small number of primitive operations
 - estimating the number of primitive operations executed up to a constant factor

```
1  def find_max(data):  
2      """Return the maximum element from a nonempty Python list."""  
3      biggest = data[0]           # The initial value to beat  
4      for val in data:           # For each value:  
5          if val > biggest        # if it is greater than the best so far,  
6              biggest = val       # we have found a new best (so far)  
7      return biggest             # When loop ends, biggest is the max
```

The “Big-Oh” Notation

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers
 - We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c * g(n), \quad \text{for } n \geq n_0$$



Properties of Big-oh

- The big-Oh notation allows us to ignore constant factors and lower-order terms and focus on the main components of a function that affect its growth
 - The highest-degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial
 - If $f(n)$ is a polynomial of degree d , that is,

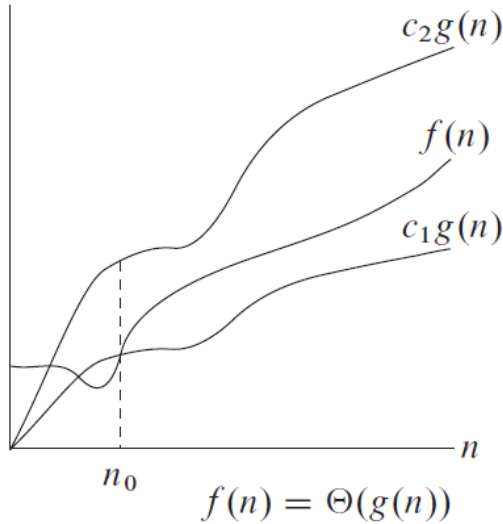
$$f(n) = a_0 + a_1n + \cdots + a_dn^d$$

- and $a_d > 0$, then $f(n)$ is $O(n^d)$

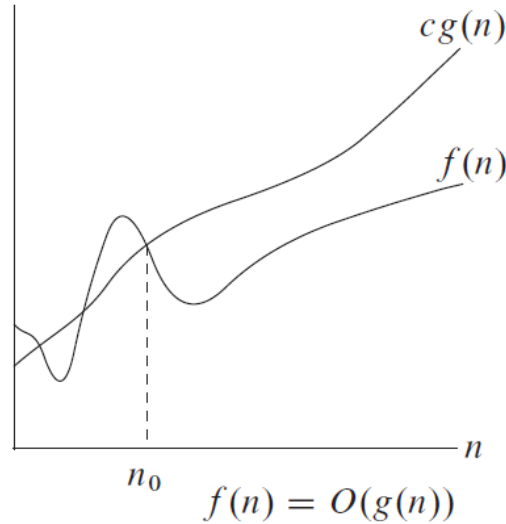
$$a_0 + a_1n + a_2n^2 + \cdots + a_dn^d \leq (|a_0| + |a_1| + |a_2| + \cdots + |a_d|)n^d.$$

We show that $f(n)$ is $O(n^d)$ by defining $c = |a_0| + |a_1| + \cdots + |a_d|$ and $n_0 = 1$.

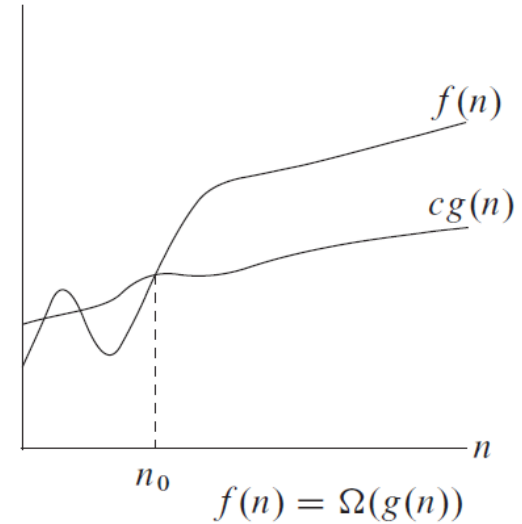
A Quicker Look at Big-omega and Theta



Tight Bound



Upper Bound



Lower Bound

Problems based on Big-oh

1. Prove that $2n^2 = O(n^3)$.

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0 \quad \text{-- 1}$$

In the given problem, $f(n) = 2n^2$ and $g(n) = n^3$

Let's now substitute $f(n)$ and $g(n)$ in eq. 1

$$2n^2 \leq c \cdot n^3$$

After cancelling n^2 on both sides, $2 \leq c \cdot n$

$$c \geq \frac{2}{n}$$

From the above, $c = 2$ and $n = 1$. Hence, $2n^2$ is $O(n^3)$, for all $n \geq 1$

Problems based on Big-oh

2. Prove that $1000n^2 + 1000n = O(n^2)$

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0 \quad \text{-- 1}$$

In the given problem, $f(n) = 1000n^2 + 1000n$ and $g(n) = n^2$

Let's now substitute $f(n)$ and $g(n)$ in eq. 1

$$1000n^2 + 1000n \leq c \cdot n^2$$

$$1000n(n + 1) \leq c \cdot n^2$$

$$1000(n + 1) \leq c \cdot n$$

$$c \geq \frac{1000(n + 1)}{n}$$

From the above, $c = 2000$ and $n = 1$. Hence, $1000n^2 + 1000n$ is $O(n^2)$, for all $n \geq 1$

Asymptotic Analysis

3. Prove that n^3 is $O(n^2)$.

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0 \quad \text{-- 1}$$

In the given problem, $f(n) = n^3$ and $g(n) = n^2$

Let's now substitute $f(n)$ and $g(n)$ in eq. 1

$$n^3 \leq c \cdot n^2$$

$$n \leq c$$

$$c \geq n$$

Hence, n^3 is $O(n^2)$ is not possible.

Problems based on Big-oh

4. Prove that $5n^2 + 3n\log n + 2n + 5 = O(n^2)$

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0 \quad \text{-- 1}$$

In the given problem, $f(n) = 5n^2 + 3n\log n + 2n + 5$ and $g(n) = n^2$

Let's now substitute $f(n)$ and $g(n)$ in eq. 1

$$5n^2 + 3n\log n + 2n + 5 \leq c \cdot n^2 \quad \text{-- 2}$$

Replace all the terms in eq. 2 by the term with maximum degree.

$$5n^2 + 3n^2 + 2n^2 + 5n^2 \leq c \cdot n^2$$

$$15n^2 \leq c \cdot n^2$$

$$c \geq 15, \text{ when } n = 1$$

Hence, $5n^2 + 3n\log n + 2n + 5$ is $O(n^2)$ is true.

Exercise Problems

1. Prove that $20n^3 + 3n\log n + 2n + 5$ is $O(n^2)$.

3-- Let $f(n) = 2n+2$ and $g(n) = 2n$. We want to show that there exists a constant c such that $f(n) \leq cg(n)$ for all sufficiently large values of n .

Let $c = 3$. Then, for all $n \geq 1$, we have

$$f(n) = 2n+2 \leq 3 \cdot 2n = 6n$$

Therefore, there exists a constant c such that $f(n) \leq cg(n)$ for all sufficiently large values of n . Hence, $2n+2$ is $O(2n)$.

2. Prove that $3\log n + 2$ is $O(\log n)$.

$$2 \leq 3\log n + 2 \leq c \cdot \log n$$

$$3\log n + 2 \leq \log n$$

$$4\log n \leq \log n$$

Since c is greater than or equal to the maximum value of $3\log n + 2$ for all $n \geq 1$, we know that $4\log n \leq \log n$ for all $n \geq 1$. Therefore, $3\log n + 2$ is $O(\log n)$.

3. Prove that 2^{n+2} is $O(2^n)$.^{yes}

4. Prove that $2n + 100\log n$ is $O(n)$.

4-- Let $f(n) = 2n+100\log n$ and $g(n) = n$. We want to show that there exists a constant c such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Let $c = 101$. Then, for all $n \geq 1$, we have

$$f(n) = 2n+100\log n \leq 2n \cdot (1 + 100\log 2n) \leq 2n \cdot (1 + 100) = 2101n$$

Therefore, there exists a constant c such that $f(n) \leq c \cdot g(n)$ for all $n \geq 1$. Hence, $2n+100\log n$ is $O(n)$.

5. Prove that $2n + 10$ is $O(n)$.

5-- Let $f(n) = 2n+10$ and $g(n) = n$. We want to show that there exists a constant c such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Let $c = 11$. Then, for all $n \geq 1$, we have

$$f(n) = 2n+10 \leq 2n \cdot (1 + 10/2n) \leq 2n \cdot (1 + 11) = 13n$$

Therefore, there exists a constant c such that $f(n) \leq c \cdot g(n)$ for all $n \geq 1$. Hence, $2n+10$ is $O(n)$.

6. Prove that 2^{2n} is $O(2^n)$.

6-- Let $f(n) = 2^{2n}$ and $g(n) = 2^n$. We want to show that there exists no constant c such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Suppose there is such a constant c . Then, for all $n \geq n_0$, we have

$$f(n) = 2^{2n} \leq c \cdot g(n) = c \cdot 2^n$$

or

Problems on Asymptotic Analysis

1. Calculate the time complexity for the function given below by counting its primitive operations.

```
void prefix_average(int a, int n)
{
    int total = 0;
    int pavg[n];
    for(i=0; i<n; i++)
    {
        total = total + a[i];
        pavg[i] = total/(i+1);
        printf("The prefix average is %f", pavg[i]);
    }
}
```


Problems on Asymptotic Analysis

Instruction	Time Complexity
<code>void prefix_average(int a, int n)</code> <code>{</code> <code>total = 0;</code>	Constant Time
<code>int pavg[n];</code>	Constant Time
<code>for(i=0; i<n; i++)</code> <code>{</code> <code>total = total + a[i];</code> <code>pavg[i] = total/(i+1);</code> <code>printf("The prefix</code> <code>average is %f", pavg[i]);</code> <code>}</code> <code>}</code>	$O(n)$

Time taken by the for loop

Iteration	i
1	0
2	1
3	2
...	...
k	k-1

$$k - 1 = n - 1$$

$$k = n$$

$$O(n)$$

Problems on Asymptotic Analysis

2. Calculate the time complexity for the function given below by counting its primitive operations.

```
void fun(int n)
{
    int i, j, k, count = 0;
    for(i = n/2; i <= n; i++)
    {
        for(j = 1; j + (n/2) <= n; j++)
        {
            for(k = 1; k <= n; k = k * 2)
            {
                Count++;
            }
        }
    }
}
```

Problems on Asymptotic Analysis

Instruction	Time Complexity
void fun(int n) { int i, j, k, count =0;	Constant Time
for(i=n/2; i<=n; i++) {	O(n)
for(j=1; j+(n/2)<=n; j++) {	
for(k=1; k<=n; k=k*2) {	
Count++; } } } }	

Time taken by the for loop 1

Iteration	i
1	$(n/2) + 0$
2	$(n/2) + 1$
3	$(n/2) + 2$
...	...
k	$(n/2) + k - 1$

$$\frac{n}{2} + k - 1 = n$$

$$k - 1 = n - \frac{n}{2}$$

$$k = \frac{n}{2}$$

$$O(n)$$

Problems on Asymptotic Analysis

Instruction	Time Complexity
void fun(int n) { int i, j, k, count =0;	Constant Time
for(i=n/2; i<=n; i++) {	O(n)
for(j=1; j+(n/2)<=n; j++) {	O(n)
for(k=1; k<=n; k=k*2) {	
Count++; } } } }	

Time taken by the for loop 2

Iteration	j
1	1
2	2
3	3
...	...
k	k

$$j + \frac{n}{2} = n$$

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

$$j = \frac{n}{2}$$

$$O(n)$$

Problems on Asymptotic Analysis

Instruction	Time Complexity
void fun(int n) { int i, j, k, count =0;	Constant Time
for(i=n/2; i<=n; i++) {	$O(n)$
for(j=1; j+(n/2)<=n; j++) {	$O(n)$
for(k=1; k<=n; k=k*2) {	$O(\log n)$
Count++; } } } }	$n * n * \log n$ $O(n^2 \log n)$

Time taken by the for loop 3

Iteration	k
1	$1 = 2^0$
2	$2 = 2^1$
3	$4 = 2^2$
...	...
p	$p = 2^{p-1}$

$$2^{p-1} = n$$

$$\log 2^{p-1} = \log n$$

$$(p-1) \log 2 = \log n$$

$$p = \log n + 1$$

$$O(\log n)$$

Problems on Asymptotic Analysis – Exercise 1

3. Calculate the time complexity for the function given below by counting its primitive operations.

```
void fun(int n)
{
    if(n<=1) return;
    int i, j;
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n; j++)
        {
            printf("Hello!");
            break;
        }
    }
}
```

Problems on Asymptotic Analysis – Exercise 2

4. Calculate the time complexity for the function given below by counting its primitive operations.

```
void fun(int n)
{
    int i, j;
    for(i=1; i<=(n/3); i++)
    {
        for(j=1; j<=n; j+=4)
        {
            printf("Hello!");
        }
    }
}
```

Asymptotic Analysis

1. The number of operations executed by algorithms A and B is $8n \log n$ and $2n^2$, respectively. Determine n_0 such that A is better than B for $n \geq n_0$.

Sure. Here is the solution to determine n_0 such that A is better than B for $n \geq n_0$:

We can compare the two algorithms by setting the number of operations executed by each algorithm equal to each other and solving for n .

$$8n \log n = 2n^2$$

$$4 \log n = n$$

$$\log n = n/4$$

$$n = 4^{\log n}$$

$$n = 4^2$$

$$n = 16$$

Therefore, $n_0 = 16$. For $n \geq 16$, algorithm A will execute fewer operations than algorithm B .

Here is a more detailed explanation of the solution:

The first step is to set the number of operations executed by each algorithm equal to each other.

$$8n \log n = 2n^2$$

The next step is to solve for n .

$$4 \log n = n$$

$$\log n = n/4$$

$$n = 4^{\log n}$$

$$n = 4^2$$

$$n = 16$$

Asymptotic Analysis

- The number of operations executed by algorithms A and B is $40n^2$ and $2n^3$, respectively. Determine n_0 such that A is better than B for $n \geq n_0$.

Sure. Here is the solution to determine n_0 such that A is better than B for $n \geq n_0$:

We can compare the two algorithms by setting the number of operations executed by each algorithm equal to each other and solving for n .

$$40n^2 = 2n^3$$

$$20n^2 = n^3$$

$$n^2 = n^3/20$$

$$n = n^{3/20}$$

$$n = (2^{3/20})^{(1/2)}$$

$$n = 2^{3/40}$$

Therefore, $n_0 = 2^{3/40}$. For $n \geq 2^{3/40}$, algorithm A will execute fewer operations than algorithm B .

Here is a more detailed explanation of the solution:

The first step is to set the number of operations executed by each algorithm equal to each other.

$$40n^2 = 2n^3$$

The next step is to solve for n .

$$20n^2 = n^3$$

$$n^2 = n^3/20$$

$$n = n^{3/20}$$

$$n = (2^{3/20})^{(1/2)}$$

$$n = 2^{3/40}$$

Finally, we can conclude that for $n \geq 2^{3/40}$, algorithm A will execute fewer operations than algorithm B .

Notion of time and space complexity

- Time Complexity
 - Time complexity is the time taken by the algorithm to execute each set of instructions. It is always better to select the most efficient algorithm when a simple problem can solve with different methods
- Space Complexity
 - Space complexity is usually referred to as the amount of memory consumed by the algorithm
 - It is composed of two different spaces; Auxiliary space and Input space