CHAPTER ONE

Introduction to Data Structures and Algorithms

Introduction

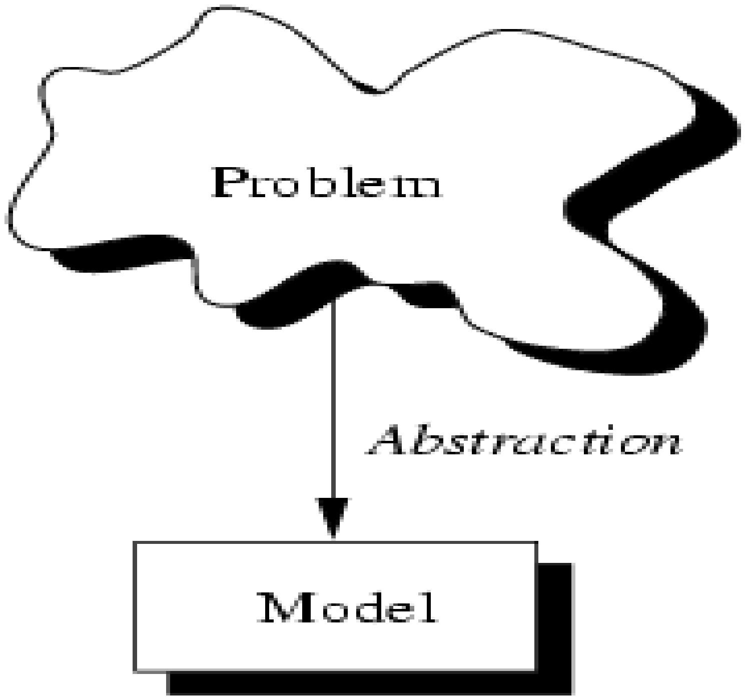
- A program is a set of instruction which is written to solve a problem.
- A solution to a problem actually consists of two things:
    - ✓ A way to organize the data
    - ✓ Sequence of steps to solve the problem
- The way data are organized in a computers memory is said to be Data Structure.
- The sequence of computational steps to solve a problem is said to be an Algorithm.
- Therefore, a *program* is Data structures plus Algorithm.

Introduction

- Data Structure is a branch of Computer Science.
- The study of data structure allows us to understand the organization of data and the management of the data flow in order to increase the efficiency of any process or program.
- A data structure is a specialized format for organizing, processing, retrieving and storing data, so that data can be used efficiently.
- Data Structures are the building blocks of any software or program.

Introduction

- Data structures are used to model the problem or part of the problem.
- The first step to solve the problem is obtaining ones own abstract view, or model, of the problem.
- This process of modeling is called abstraction.



- The model defines an abstract view to the problem.
- The model should only focus on problem related stuff.

Abstraction

- Abstraction is a process of classifying characteristics as relevant and irrelevant for the particular purpose at hand and ignoring the irrelevant ones.

Example: model students of a University.

- Relevant:

```
Char Name[15];
Char ID[11];
Char Dept[20];
int Age, year;
```

- Non relevant

```
float height, weight;
```

Abstraction

- Using the model, a programmer tries to define the properties of the problem.
- These properties include
  - ✓ The data which are affected and
  - ✓ The operations that are involved in the problem
- An entity with the properties just described is called an Abstract Data Type (ADT).

Abstract Data Types

- ADT consists of data to be stored and operations supported on them.
- ADT is a specification that describes a data set and the operation on that data.
- The ADT specifies:
  - ✓ What data is stored.
  - ✓ What operations can be done on the data.
- ADT does not specify how to store or how to implement the operation.
- ADT is independent of any programming language

Abstract Data Types

Example: ADT employees of an organization:

- This ADT stores employees with their relevant attributes and discarding irrelevant attributes.

✓ Relevant:- Name, ID, Sex, Age, Salary, Dept, Address

✓ Non Relevant :- weight, color, height

✓ This ADT supports hiring, firing, retiring, … operations.

Data Structure

- In Contrast a data structure is a language construct that the programmer has defined in order to implement an *abstract data type.*

- What is the purpose of data structures in programs?
  - Data structures are used to model a problem.

Example:

```
struct Student_Record
{
        char name[20];
        char ID_NO[10];
        char Department[10];
        int age;
};
```

Data Structure

- Attributes of each variable:
  - Name: Textual label.
  - Address: Location in memory.
  - Scope: Visibility in statements of a program.
  - Type: Set of values that can be stored and set of operations that can be performed.
  - Size: The amount of storage required to represent the variable.
  - Life time: The time interval during execution of a program while the variable exists.

Data Structure

- **Characteristics of a Data Structure**

  - Correctness: Data structure implementation should implement its interface correctly.
    - Each data structure has an interface. Interface represents the set of operations that a data structure supports.

  - Time Complexity: Running time or the execution time of operations of data structure must be as small as possible.

  - Space Complexity: Memory usage of a data structure operation should be as little as possible.

Algorithm

- Algorithm is a concise specification of an operation for solving a problem.
- Algorithm is a well-defined computational procedure that takes some value or a set of values as input and produces some value or a set of values as output.

$$\text{Inputs} \longrightarrow \text{Algorithm} \longrightarrow \text{Outputs}$$

- An algorithm is a specification of a behavioral process.
- It consists of a finite set of instructions that govern behavior step-by-step.
- It is part of what constitutes a data structure

Algorithm

- Data structures model the static part of the problem.
  - They are unchanging while the problem is changing.
- In order to model the dynamic part of the problem we need to work with algorithms.
- Algorithms are the dynamic part of a program's problem model.
- An algorithm transforms data structures from one state to another state.

Algorithm

- What is the purpose of algorithms in programs?
  - Take values as input.

    Example: `cin>>age;`
  - Change the values held by data structures.

    Example: `age=age+1;`
  - Change the organization of the data structure:

    Example: Sort students by name
  - Produce outputs:

    Example: Display student's information

Algorithm

- The quality of a data structure is related to its ability to successfully model the characteristics of the problem.
- Similarly, the quality of an algorithm is related to its ability to successfully simulate the changes in the problem.
- However, the quality of data structure and algorithms is determined by their ability to work together well.
- Generally speaking, correct data structures lead to simple and efficient algorithms and correct algorithms lead to accurate and efficient data structures.

Properties of Algorithms

1. Finiteness:

- Algorithm must complete after a finite number of steps.
- Algorithm should have a finite number of steps.

Example of Finite Steps

```
int i=0;
while(i<10)
{
  cout<< i;
  i++;
}
```

Example of Infinite Steps

```
while(true)
{
  cout<<"Hello";
}
```

Properties of Algorithms

2. Definiteness (Absence of ambiguity):

- Each step must be clearly defined, having one and only one interpretation.
- At each point in computation, one should be able to tell exactly what happens next.

3. Sequential:

- Each step must have a uniquely defined preceding and succeeding step.
- The first step (start step) and last step (halt step) must be clearly noted.

Properties of Algorithms

4. Feasibility:

- It must be possible to perform each instruction.
- Each instruction should have possibility to be executed.

a)
```
for(int i=0; i<0; i++)
   {
     cout<< i;  // there is no possibility that this
    }            //statement to be executed.
```

b)
```
if(5>7)
{
   cout<<"hello"; // not executed.
}
```

Properties of Algorithms

5. Correctness:
    - It must compute correct answer for all possible legal inputs.
    - The output should be as expected and required and correct.
6. Language Independence:
    - It must not depend on anyone programming language.
7. Completeness:
    - It must solve the problem completely.

Properties of Algorithms

8. Effectiveness:
   - Doing the right thing. It should return the correct result all the time for all of the possible cases.

9. Efficiency:
   - It must solve with the least amount of computational resources such as time and space.
   - Producing an output as per the requirement within the given resources (constraints).

Example: Write a program that takes a number and displays the square of the number.

1)
```
int x;
cin>>x;
cout<<x*x;
```

2)
```
int x,y;
cin>>x;
y=x*x;
cout<<y;
```

Properties of Algorithms

Example: Write a program that takes two numbers and displays the sum of the two.

Program a
```
cin>>a;
cin>>b;
sum= a+b;
cout<<sum;
```

Program b
```
cin>>a;
cin>>b;
a= a+b;
cout<<a;
```

Program c (the most efficient)
```
cin>>a;
cin>>b;
cout<<a+b;
```

All are effective but with different efficiencies.

Properties of Algorithms

10. Input/output:

- There must be a specified number of input values, and one or more result values.
- Zero or more inputs and one or more outputs.

11. Precision:

- The result should always be the same if the algorithm is given identical input.

12. Simplicity:

- A good general rule is that each step should carry out one logical step.
  - What is simple to one processor may not be simple to another.

Algorithm Analysis

- Algorithm analysis refers to the process of determining how much computing time and storage that algorithms will require.

- In other words, it's a process of predicting the resource requirement of algorithms in a given environment.

- In order to solve a problem, there are many possible algorithms.

- One must be able to choose the best algorithm for the problem at hand using some scientific method.

- To classify some data structures and algorithms as good:
  - we need precise ways of analyzing them in terms of resource requirement.

  The main resources are:
    - Running Time
    - Memory Usage

- Note: **Running time** is the most important since computational time is the most precious resource in most problem domains.

- There are *two* approaches to measure the efficiency of algorithms: Empirical and Theoretical

1. Empirical
   - Based on the total running time of the program.
   - Uses actual system clock time.

- Example:

t1= initial time before the program starts

```
for(int i=0; i<=10; i++)
    cout<<i;
```

t2 = final time after the execution of the program is finished

Running time taken by the above algorithm or

Total Time = t2-t1;

- It is difficult to determine efficiency of algorithms using Empirical approach, because clock-time can vary based on many factors.

For example:

a) Processor speed of the computer

1.78GHz	2.12GHz
10s	<10s

b) Current processor load

- Only the work 10s
- With printing 15s
- With printing & browsing the internet >15s

c) Specific data for a particular run of the program

- Input size, Input properties

```
t1
  for(int i=0; i<=n; i++)
      cout<<i;
t2
T=t2-t1;
```

For n=100, T>=0.5s

  n=1000, T>0.5s

d) Operating System

- Multitasking Vs Single tasking

2. Theoretical

- Determining the quantity of resources required mathematically (Execution time, memory space, etc.) needed by each algorithm.

- Analyze an algorithm according to the number of basic operations (time units) required, rather than according to an absolute amount of time involved.

2. Theoretical

- We use theoretical approach to determine the efficiency of algorithm because:
  - The number of operation will not vary under different conditions.
  - It helps us to have a meaningful measure that permits comparison of algorithms independent of operating platform.
  - It helps to determine the complexity of algorithm.

Complexity Analysis

- Complexity Analysis is the systematic study of the cost of computation, measured either in:
  - Time units
  - Operations performed, or
  - The amount of storage space required.

- Two important ways to characterize the effectiveness of an algorithm are its:
  - Space Complexity and
  - Time Complexity

Time Complexity

- The amount of time required to complete the execution.
- Determine the approximate amount of time (number of operations) required to solve a problem of size n.
- The limiting behavior of time complexity as size increases is called the Asymptotic Time Complexity.

```
sum=0;
// Suppose we have to calculate the sum of n numbers.
for i=1 to n
sum=sum+i;
// when the loop ends then sum holds the sum of the n numbers
return sum;
```

- In the sample code, the time complexity of the loop statement will be at least n, and if the value of n increases, then the time complexity also increases.
- We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

Space Complexity

- The amount of space required to solve a problem and produce an output.
- Determine the approximate memory required to solve a problem of size n.
- The limiting behavior of space complexity as size increases is called the Asymptotic Space Complexity.

- For an algorithm, the space is required for the following purposes:
  - To store program instructions
  - To store constant values
  - To store variable values
  - To track the function calls, jumping statements, etc.

- Asymptotic Complexity of an algorithm determines the size of problems that can be solved by the algorithm.

- Factors affecting the running time of a program:
  - ✓ CPU type (80286, 80386, 80486, Pentium I---IV)
  - ✓ Memory used
  - ✓ Computer used,
  - ✓ Programming Language - C (fastest), C++ (faster), Java (fast)
  - ✓ Algorithm used and Input size

- Note: Important factors for this course are Input size and Algorithm used.

- Complexity analysis involves two distinct phases:

1. Algorithm Analysis:

- Analysis of the algorithm or data structure to produce a function T(n) that describes the algorithm in terms of the operations performed in order to measure the complexity of the algorithm.
- Example: Suppose we have hardware capable of executing 10^6 instructions per second. How long would it take to execute an algorithm whose complexity function is $T(n)=2n^2$ on an input size of $n=10^8$?

- Solution:
  - $T(n)= 2n^2=2(10^8)^2 = 2*10^{16}$
  - Running time$=T(10^8)/10^6=2*10^{16}/10^6=2*10^{10}$ seconds.

2. Order of Magnitude Analysis:

- Analysis of the function *T(n)* to determine the general complexity category to which it belongs.

- There is no generally accepted set of rules for algorithm analysis.
- However, an exact count of operations is commonly used.
- To count the number of operations we can use the following Analysis Rule.

Analysis Rules

1. Assume an arbitrary time unit.
2. Execution of one of the following operations takes time 1 unit:

   - Assignment Operation, Example: i=0;
   - Single Input/Output Operation, Example: cin>>a; and cout<<"hello";
   - Single Boolean Operations, Example: i>=10;
   - Single Arithmetic Operations, Example: a+b;
   - Function Return, Example: return sum;

3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.

Analysis Rules

Example 1:

```
int x, y;
int sum=0;
sum = x +y;
return sum;
```

T(n) = 1 + 1 + 1 + 1 = 4

Example 2:

```
int x;
int sum=0;
if(a>b)
   {
     sum= a+b;
     cout<<sum;
   }
else
   {
     cout<<b;
   }
```

$\boxed{T(n) = 1 + 1 + \max(3,1) = 5}$

Analysis Rules

4. Loop statements:

- The running time for the statements inside the loop * number of iterations + time for setup (1) + time for checking (number of iteration + 1) + time for update (number of iteration).
- The total running time of statements inside a group of nested loops is the running time of the statements * the product of the sizes of all the loops.
- For nested loops, analyze inside out.
- Always assume that the loop executes the maximum number of iterations possible. (Why?)
    - because we are interested in the worst case complexity.

Analysis Rules

Examples:

1a)

```
int k=0, n;
cout<<"Enter an integer";
cin>>n
for(int i=0; i<n; i++)
        k++;
```

$T(n) = 1+1+1+(1+n+1+n)+n$
$T(n) = 3n+5$

1b)

```
int k=0, n;
cout<<"Enter an integer";
cin>>n
for(int i=0; i<n; i++)
        k=k+1;
```

$T(n) = 1+1+1+(1+n+1+n)+(1+1)n$
$T(n) = 4n+5$

Analysis Rules

5. Function call:

- 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.

Examples:

```
int total(int n)
   {
        int sum=0;
        for (int i=1; i<=n; i++)
         sum=sum+1;
        return sum;
    }
```

$$T(n) = 1+ (1+n+1+n)+(1+1)n+1$$
$$T(n) = 4n+4$$

Analysis Rules

Example:

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

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

$= 2n+3+n(3n+2)$

$= 2n+3+3n^2+2n$

$= 3n^2+4n+3$

Class Activity

1).
```
int counter()
  {
    int a=0;
    cout<<"Enter a number";

    cin>>n;
    for(i=0; i<n; i++)
       a=a+1;
     return 0;
  }
```

$T(n)=1+1+1+(1+n+1+n)+2n+1$
$=4n+6$

2).
```
int i=0;
while(i<n)
{
   cout<<i;
   i++;
 }
  int j=1;
  while(j<=10)
{
  cout<<j;
  j++;
}
```

$T(n) = 1+n+1+n+n+1+11+10+10$
$T(n) = 3n+34$

Class Activity

```
3). void func( ){
        int x=0;  int i=0;  int j=1;
        cout<<"Enter a number";
        cin>>n;
        while(i<n){
    i=i+1;
      }
        while(j<n){
            j=j+1;
        }
     }
```

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

$= 6+4n+2n-2$

$=6n+4$

Class Activity

4a).

```
int sum(int n)
{
    int s=0;
   for(int i=1; i<=n; i++)
        s=s+(i*i*i*i);

        return s;
 }
```

$T(n)=1+(1+n+1+n+5n)+1$
$=7n+4$

4b).

```
int sum(int n)
{
    int s=0;
    for(int i=1; i<n; i++)
        s=s+(i*i*i*i);

        return s;
 }
```

$T(n)=1+(1+n+(n-1)+5(n-1))+1$
$=1+(2n+5n-5)+1$
$=7n-3$

Formal Approach to Analysis

- In the above examples we have seen that analyzing Loop statements is so complex.
- It can be simplified by using some formal approach in which case we can ignore initializations, loop controls, and updates.

Simple Loops: Formally

- For loop can be translated to a summation.
- The index and bounds of the summation are the same as the index and bounds of the for loop.

- Suppose we count the number of additions that are done.
- There is 1 addition per iteration of the loop, hence n additions in total.

```
for (int i = 1; i <= N; i++) {
     sum = sum+i;
}
```

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

Nested Loops: Formally

- Nested for loops translate into multiple summations, one for each For loop.

```
for (int i = 1; i <= N; i++) {
     for (int j = 1; j <= M; j++) {
          sum = sum+i+j;
     }
}
```

$$\sum_{i=1}^{N}\sum_{j=1}^{M} 2 = \sum_{i=1}^{N} 2M = 2MN$$

Consecutive Statements: Formally

- Add the running times of the separate blocks of your code.

```
for (int i = 1; i <= N; i++) {
      sum = sum+i;
}
for (int i = 1; i <= N; i++) {
      for (int j = 1; j <= N; j++) {
              sum = sum+i+j;
      }
}
```

$$\left[\sum_{i=1}^{N} 1\right] + \left[\sum_{i=1}^{N}\sum_{j=1}^{N} 2\right] = N + 2N^2$$

Conditionals Statements: Formally: - take maximum

```
if (test == 1) {
    for (int i = 1; i <= N; i++) {
         sum = sum+i;
}}
else for (int i = 1; i <= N; i++) {
          for (int j = 1; j <= N; j++) {
                sum = sum+i+j;
}}
```

$$\max\left(\sum_{i=1}^{N} 1, \sum_{i=1}^{N}\sum_{j=1}^{N} 2\right) =$$

$$\max\left(N, 2N^2\right) = 2N^2$$

Categories of Algorithm Analysis

- Algorithms may be examined under different situations to correctly determine their efficiency for accurate comparison.

1. Best Case Analysis:

- Requires a minimum time for program execution.
- Assumes the input data are arranged in the most advantageous order for the algorithm.
- Takes the smallest possible set of inputs.
- Causes execution of the fewest number of statements.
- Computes the lower bound of T(n), where T(n) is the complexity function.

1. Best Case Analysis: Examples:

- For sorting algorithm
  - If the list is already sorted (data are arranged in the required order).

- For searching algorithm
  - If the desired item is located at first accessed position.

2. Worst Case Analysis:

- Requires a maximum time for program execution.
- Assumes the input data are arranged in the most disadvantageous order for the algorithm.
- Takes the worst possible set of inputs.
- Causes execution of the largest number of statements (operations).
- Computes the upper bound of T(n) where T(n) is the complexity function.

Example:

- While sorting, if the list is in opposite order.
- While searching, if the desired item is located at the last position or is missing.

2. Worst Case Analysis:

- Worst case analysis is the most common analysis because it provides the upper bound for all input (even for bad ones).
- If situations are in their best case, no need to develop algorithms because data arrangements are in the best situation.
- Best case analysis can not be used to estimate complexity.
- We are interested in the worst case time since it provides a bound for all input, this is called the "Big-Oh" estimate.

3. Average Case Analysis:

- Requires an average time for program execution.
- Determine the average of the running time overall permutation of input data.
- Takes an average set of inputs.
- It also assumes random input size.
- It causes average number of executions.
- Computes the optimal bound of T(n) where T(n) is the complexity function.

- Sometimes average cases are as bad as worst cases and as good as best cases.
- Average case analysis is often difficult to determine and define.

3. Average Case Analysis: Examples:

- For sorting algorithms
    - While sorting, considering any arrangement (order of input data).
- For searching algorithms
    - While searching, if the desired item is located at any location or is missing.

Order of Magnitude

- Order of Magnitude refers to the rate at which the storage or running time grows as a function of problem size grows.
- It is expressed in terms of its relationship to some known functions.
- This type of analysis is called Asymptotic analysis.
  - Asymptotic Analysis is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound!.
  - i.e. the running time of an algorithm increases when the size of the input increases.

Asymptotic Notations

- Asymptotic Analysis makes use of O (Big-Oh) , Ω (Big-Omega), θ (Theta), o (little-o), ω (little-omega) - notations in performance analysis and characterizing the complexity of an algorithm.
- Note: The complexity of an algorithm is a numerical function of the size of the problem (instance or input size).

Types of Asymptotic Notations

1. Big-Oh Notation (upper bound)

- Definition: We say $f(n)=O(g(n))$, if there are two positive constants n_o and c, such that to the right of n_0, the value of f(n) always lies on or below $c.g(n)$.
  - i.e., $f(n) \leq c.g(n)$

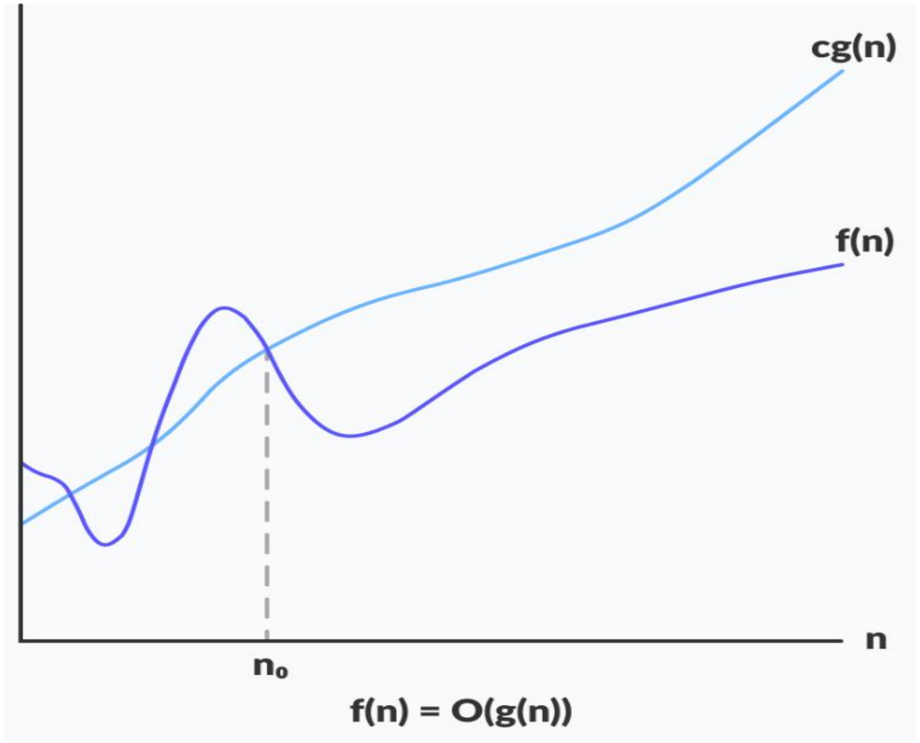
- It is the formal way to express the upper boundary of an algorithm running time.

- It's only concerned with what happens for very large values of n.

- It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation.

1. Big-Oh Notation (upper bound)

- If $f(n)$ and $g(n)$ are the two functions defined for positive integers, then $f(n) = O(g(n))$ as *f(n) is big oh of g(n)* or *f(n) is on the order of g(n)* if there exists two constants c and n_0 such that: $f(n) \leq c.g(n)$ for all $n \geq n_0$.
- This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n).



1. Big-Oh Notation: Example 1

- Given: $f(n)=10n+5$ and $g(n)=n$.
- Show that $f(n)$ is $O(g(n))$.
  - To show that $f(n)$ is $O(g(n))$, we must show that there exist two constants c and n_0 such that $f(n)<=c.g(n)$ for all $n>= n_0$.
  - $10n+5 <= c.g(n)$; for all $n>=n_0$
  - let $c=15$, then show that $10n+5<=15n$
  - $5<=5n$ or $1<=n$
  - So, $f(n)=10n+5 <= 15.g(n)$ for all $n>=1$
  - ($c=15$, $n_0=1$), there exist two constants that satisfy the above constraints.
  - Therefore, $f(n) = O(g(n))$.

1. Big-Oh Notation: Example 2

- Show that $f(n)=O(g(n))$ for $f(n)=3n^2+4n+1$ and $g(n) = n^2$.

$3n^2 <= 3n^2$ for all $n>=1$

$4n<=4n^2$ for all $n>=1$ and

$1<=n^2$ for all $n>=1$

$3n^2+4n+1<=3n^2+4n^2+n^2$ for all $n>=1$

$3n^2+4n+1<=8n^2$ for all $n>=1$

So, we have shown that $f(n)<=8n^2$ for all $n>=1$.

- Therefore, $f(n) = O(g(n))$, $(c=8, n_0=1)$, there exist two constants that satisfy the constraints.

Class Activity

Show that $f(n)=O(g(n))$ for $f(n)=2n+3$ and $g(n) = n$.

Solution:

Now, we have to find is $f(n)=O(g(n))$?

To check $f(n)=O(g(n))$, it must satisfy the given condition:

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

First, we will replace $f(n)$ by $2n+3$ and $g(n)$ by n.

$2n+3 <= c.n$

Let's assume $c=5$, then

$2n+3<=5n$

$3<=3n => 1<=n$

So, we have shown that $f(n)<=c.g(n)$ for all $n>=1$.

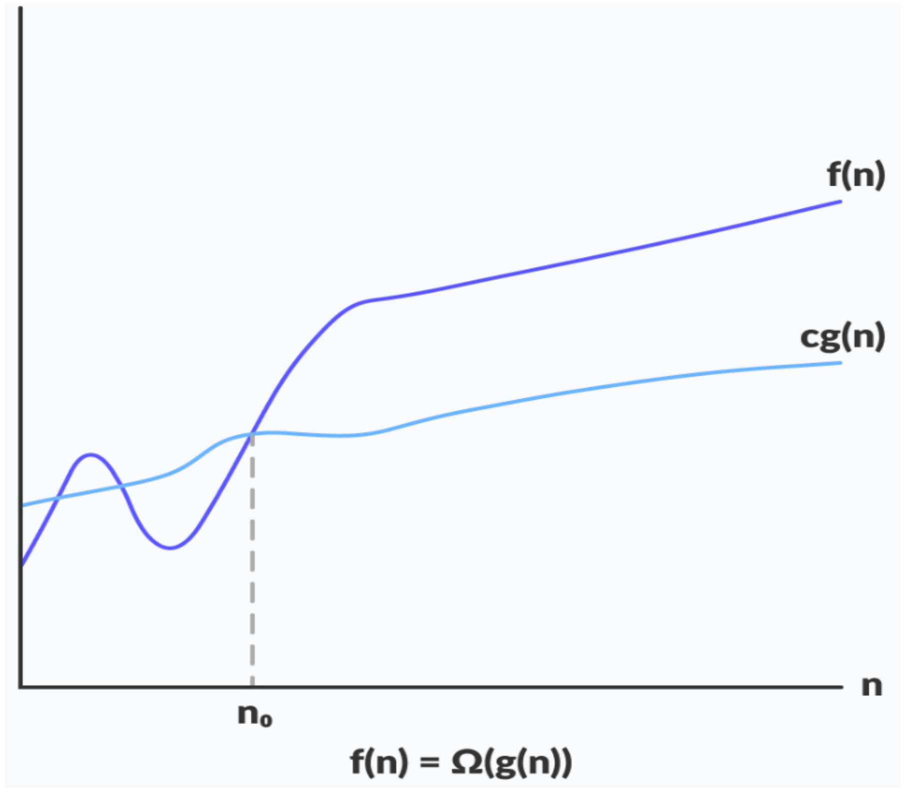
Therefore, $f(n)$ is $O(g(n))$, ($c=5$, $n_0 =1$), there exist two constants that satisfy the constraints.

2. Big-Omega (Ω)-Notation (Lower bound)

- **Definition:** We write $f(n) = \Omega(g(n))$ if there are two positive constants n_0 and c such that to the right of n_0 the value of f(n) always lies on or above c.g(n).
- It basically describes the best-case scenario which is opposite to the big O notation.
- It is the formal way to represent the lower bound of an algorithm's running time.
- It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
- It determines what is the fastest time that an algorithm can run.

2. Big-Omega (Ω)-Notation (Lower bound)

- If f(n) and g(n) are the two functions defined for positive integers, then $f(n) = \Omega(g(n))$ as f(n) is Omega of g(n) or f(n) is on the order of g(n)) if there exists two constants c and n_o such that: $f(n) >= c.g(n)$ for all $n \geq n_o$ and $c>0$.



Big-Omega (Ω)-Notation: Example 1

- ❖ Let's consider a simple example.
- ❖ If $f(n) = 2n+3$, $g(n) = n$, Is $f(n)= \Omega(g(n))$?

Solution:

- It must satisfy the condition: $f(n)>=c.g(n)$
- To check the above condition, we first replace $f(n)$ by $2n+3$ and $g(n)$ by n.
- $2n+3>=c*n$. Suppose $c=1$
- $2n+3>=n$ (This equation will be true for any value of n starting from 1).

- ❖ Therefore, it is proved that f(n) is big omega of g(n) function
  - – i.e. $f(n)$ is $\Omega(g(n))$, ($c=1$, $n_o=1$), there exist two constants that satisfy the constraints.

Big-Omega (Ω)-Notation: Example 2

❖ If $f(n) = 3n+5$ and $g(n) = \sqrt{n}$, Is $f(n)= \Omega(g(n))$?

Solution:

- It must satisfy the condition: $f(n)>=c.g(n)$
- To check the above condition, we first replace $f(n)$ by $3n+5$ and $g(n)$ by $\sqrt{n}$,
- $3n+5>=c*\sqrt{n}$. Suppose $c=1$
- $3n+5>=\sqrt{n}$, (This equation will be true for any value of n starting from 1).

❖ Therefore, it is proved that f(n) is big omega of g(n) function
  - i.e. $f(n)$ is $\Omega(g(n))$, ($c=1$, $n_o=1$), there exist two constants that satisfy the constraints.

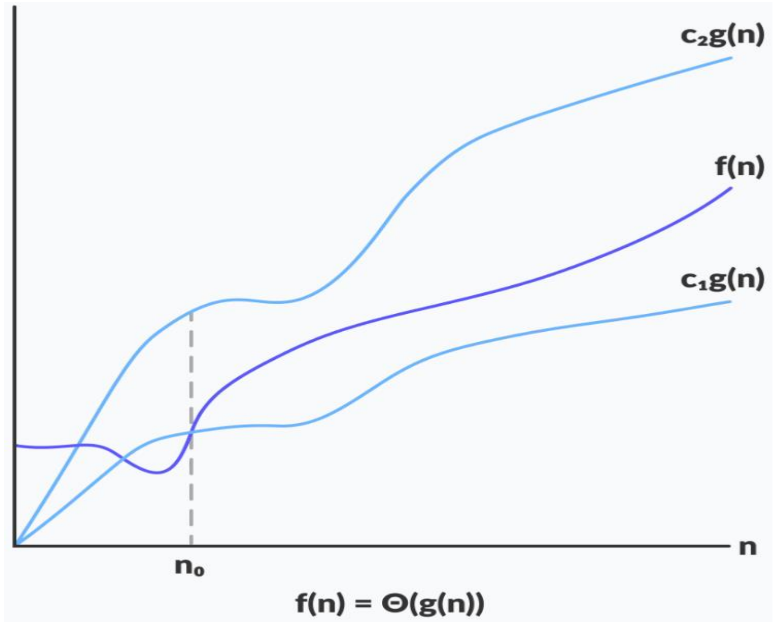
3. Theta Notation (θ-Notation) (Optimal bound)

❖ Definition: We say $f(n) = \theta(g(n))$ if there exist positive constants n_o, $c1$ and $c2$ such that to the right of n_o, the value of $f(n)$ always lies between $c1.g(n)$ and $c2.g(n)$ inclusive, i.e., $c1.g(n) <= f(n) <= c2.g(n)$, for all $n >= n_o$.

- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm.
- Big theta is mainly used when the value of worst-case and the best-case is same.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.

3. Theta Notation (θ-Notation) (Optimal bound)

- Let f(n) and g(n) be the functions of n, then:
  - $f(n) = \theta(g(n))$
- The above condition is satisfied only if when
  - $c1.g(n) <= f(n) <= c2.g(n)$
- where the function is bounded by two limits, i.e., upper and lower limit, and f(n) comes in between.



Theta Notation (θ-Notation): Example 1

- Let's consider the same example where f(n)=2n+3 and g(n)=n.
- As c1.g(n) should be less than f(n) so c1 has to be 1 whereas c2.g(n) should be greater than f(n) so c2 is equal to 5.
- Then, c1.g(n) is the lower limit of the of the f(n) while c2.g(n) is the upper limit of the f(n).
  - c1.g(n)<=f(n)<=c2.g(n)
- Replace g(n) by n, f(n) by 2n+3, c1 by 1 and c2 by 5.
  - 1.n<=2n+3<=5.n
  - n<=2n+3 <=5n, for all n>=n_0
- Therefore, we can say that for any value of n, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n). Hence, it is proved that f(n) is big theta of g(n).

Theta Notation (θ-Notation): Example 2

- Show that $f(n) = \theta(g(n))$, where $f(n)=2n^2+3$ and $g(n)=n^2$
  - As c1.g(n) should be less than f(n) so c1 has to be 1 whereas c2.g(n) should be greater than f(n) so c2 is equal to 5.
  - Then, c1.g(n) is the lower limit of the of the f(n) while c2.g(n) is the upper limit of the f(n).
    - $c1.g(n) <= f(n) <= c2.g(n)$
  - Replace g(n) by n, f(n) by 2n+3, c1 by 1 and c2 by 5.
    - $1.n^2 <= 2n^2+3 <= 5.n^2$, => $n^2 <= 2n^2+3 <= 5n^2$, for all $n >= n_0$
  - Therefore, we can say that for any value of n, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n). Hence, it is proved that f(n) is big theta of g(n).

4. Little-oh (small-oh) Notation

- Definition: We say $f(n)=o(g(n))$, if there are two positive constants n_o and c such that to the right of n_o, the value of $f(n)$ lies below $c.g(n)$.

  i.e. $f(n) < c.g(n)$, for all $n>n_0$

- As n increases, g(n) grows strictly faster than f(n).
- Describes the worst case analysis.
- Denotes an upper bound that is not asymptotically tight.
- Big O-Notation denotes an upper bound that may or may not be asymptotically tight.

Little-oh (small-oh) Notation: Example

- Show that $f(n) = o(g(n))$ for $f(n) = n^2$ and $g(n)=n^2$.
  - $f(n) < c.g(n)$, suppose $c=2$.
  - $n^2<2n^2$, for all $n>n_0$, ➔ $n_0=1$, $c=2$,
- Therefore, it is proved that f(n) is little-oh of g(n) function.
  - i.e. f(n) is o(g(n)), ($c=2$, $n_0=1$), there exist two constants that satisfy the constraints.

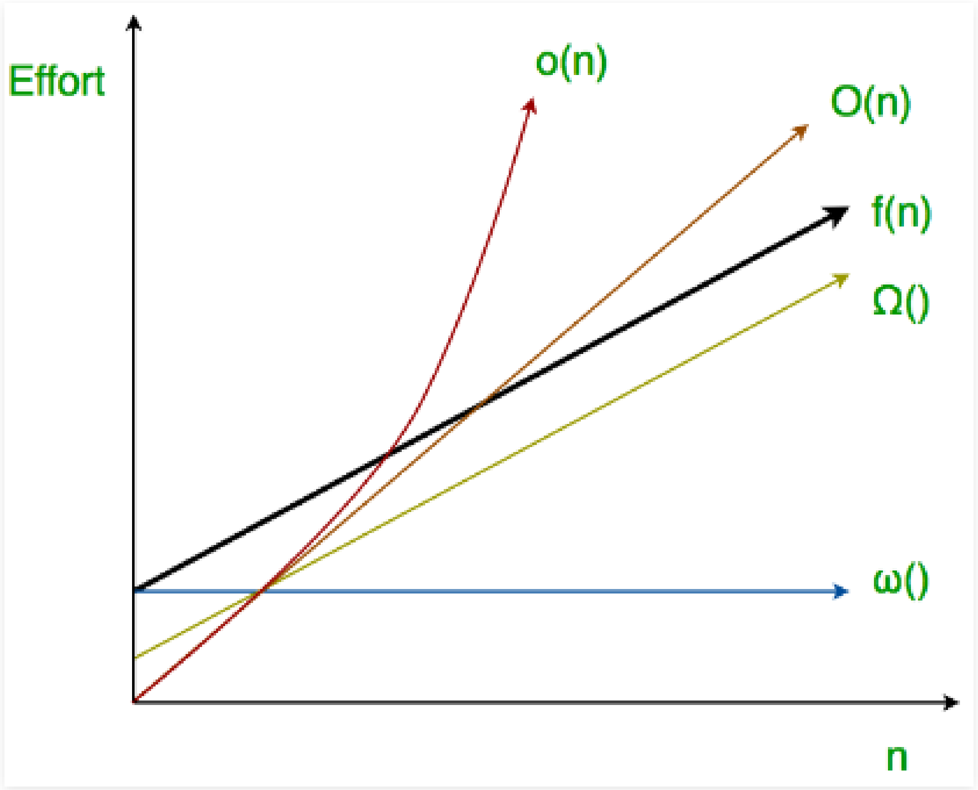
- It is the same for all of the following functions.
  - $n^2< n^3$, $g(n) = n^3$, $f(n)=o(n^3)$
  - $n^2< n^4$, $g(n) =n^4$, $f(n)=o(n^4)$

5. Little-Omega (ω) notation

❖ Definition: We write $f(n)=\omega(g(n))$, if there are positive constants n_o and c such that to the right of n_o, the value of f(n) always lies above c.g(n).

i.e. $f(n) > c.g(n)$, for all $n>n_0$

- As n increases f(n) grows strictly faster than g(n).
- Describes the best case analysis.
- Denotes a lower bound that is not asymptotically tight.
- Big Ω-Notation denotes a lower bound that may or may not be asymptotically tight.



Little-Omega (ω) notation: Example

- Find g(n) such that $f(n)=\omega(g(n))$ for $f(n)=n^2+3$ and $g(n)=n$.
  - Since $n^2 > \sqrt{n}$, $c=1$, $n_0=1$, can also be solution.

❖Rules to estimate Big Oh of a given function

1. Pick the highest order.
2. Ignore the coefficient.

Example:

1. $T(n)=3n + 5 \rightarrow O(n)$
2. $T(n)=3n^2+4n+2 \rightarrow O(n^2)$

- Some known functions encountered when analyzing algorithms. (Complexity category for Big-Oh).

See next slide ➔

Rule 1:

- If T1(n)=O(f(n)) and T2(n)=O(g(n)), then
  - ✓ T1(n)+T2(n)=max(O(f(n)),O(g(n)))
  - ✓ T1(n)*T2(n)=O(f(n)*g(n))

Rule 2:

- If T(n) is a polynomial of degree k, then $T(n)=\theta(n^k)$.

Rule 3:

- lognk=O(n) for any constant k. This tells us that logarithms grow very slowly.
- We can always determine the relative growth rates of two functions f(n) and g(n) by computing

  limn → infinity f(n)/g(n).
- The limit can have four possible values.

- The limit is 0: This means that $f(n)=o(g(n))$.
- The limit is $c \neq 0$: This means that $f(n)=\theta(g(n))$.
- The limit is infinity: This means that $g(n)=O(f(n))$.
- The limit oscillates: This means that there is no relation between $f(n)$ and $g(n)$.

Example:

- n^3 grows faster than n^2, so we can say that $n^2=O(n^3)$ or $n^3=\Omega(n^2)$.
- $f(n)=n^2$ and $g(n)=2n^2$ grow at the same rate, so both $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$ are true.
- If $f(n)=2n^2$, $f(n)=O(n^4)$, $f(n)=O(n^3)$, and $f(n)=O(n^2)$ are all correct, but the last option is the best answer.

$T(n)$	Complexity Category functions $F(n)$	Big-O
c, c is constant	1	$C=O(1)$
$10\log n + 5$	$\log n$	$T(n)=O(\log n)$
$\sqrt{n} +2$	$\sqrt{n}$	$T(n)=O(\sqrt{n})$
$5n+3$	n	$T(n)=O(n)$
$3n\log n+5n+2$	$n\log n$	$T(n)=O(n\log n)$
$10n^2 +n\log n+1$	n^2	$T(n)=O(n^2)$
$5n^3 + 2n^2 + 5$	n^3	$T(n)=O(n^3)$
2^n+n^5+n+1	2^n	$T(n)=O(2^n)$
$7n!+2^n+n^2+1$	$n!$	$T(n)=O(n!)$
$8n^n+2^n +n^2 +3$	n^n	$T(n)=O(n^n)$

Complexity category	**Big-Oh**	**Example**
Constant	$T(n)=O(1)$-constant growth	Determining if a number is even or odd.
Logarithmic	$T(n)=O(\log n)$	Finding an item in a sorted array with a binary search. $10\log n+5$
Linear	$T(n)=O(n)$	Finding an item in an unsorted list, adding two n-digit numbers. $5/3n+10$
Loglinear	$T(n)=O(n\log n)$	Merge sort, heap sort $3n\log n+5n+2$
Quadratic	$T(n)=O(n^2)$	Adding two nXn matrices, shell sort, insertion sort, multiplying two n-digit numbers by a simple algorithm. $11/7n^2+2n+5$
Cubic	$T(n)=O(n^3)$	Multiplying two nXn matrices by simple algorithm. $3n^3+4n^2+5n+7$
Exponential	$T(n)=O(c^n)$, $c>1$	2^n+n^2+n+1
Factorial	$T(n)=O(n!)$	$7n!+n^2+1$
Double exponential	$T(n)=O(c1^{c2n})$, c1 and c2>1	$2^{2n}+2^n+n^2+1$
n^n	$T(n)=O(n^n)$	$8n^n+2^n+n^2+3$

- Arrangement of common functions by growth rate.
- List of typical growth rates.

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	Log-Linear
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Class Activity

- The order of the body statements of a given algorithm is very important in determining Big-Oh of the algorithm.

Example: Find Big-Oh of the following algorithm.

1.
```
for( int i=1;i<=n; i++)
    sum=sum + i;
```
$T(n)=2*n=2n=O(n)$.

2.
```
for(int i=1; i<=n; i++)
    for(int j=1; j<=n; j++)
        k++;
```
$T(n)=1*n*n=n^2 = O(n^2)$.

Thank You

Question?