SCS1020 Foundations of Algorithm

Dr. Dinuni Fernando
PhD

Senior Lecturer

BINGHAMTON UNIVERSITY
STATE UNIVERSITY OF NEW YORK

Dr. Dinuni K. Fernando

- BSc.(Hons) - University of Colombo School of Computing class of 2014 - (2009-2014) - CS major
- Ph.D - State University of New York at Binghamton, 2019
  - Articles and journals in top-tier prestigious international conferences and journals.
  - Groundbreaking work has earned a US patent.
  - Heads the CloudNet research group

Research Interests

- Cloud Computing / Virtualization
- Fault tolerance, ML inspired Cloud Computing
- Blockchain and Security

Learning Objectives

LO1 : Understand algorithm efficiency: Students will be able to define and explain the concept of efficiency in computer programs and its importance in algorithm design.

LO2 : Analyze algorithms: Students will demonstrate the ability to apply standard analysis methods (e.g., time and space complexity) to evaluate and compare the performance of algorithms, particularly focusing on searching and sorting problems.

LO3 : Classify algorithms by complexity: Students will categorize algorithms into various complexity classes (e.g., constant, logarithmic, linear, quadratic) based on their performance analysis.

Learning Objectives

LO4 : Learn and apply tree data structures: Students will explore different tree data structures (e.g., binary search trees, AVL trees, heaps) and describe their design, operations, and use cases for efficient data organization and retrieval.

LO5 : Evaluate efficiency in tree structures: Students will analyze the efficiency of tree data structures and variations, considering the underlying requirements for time and space efficiency in specific applications.

Course Structure

- Lecture Tuesday 8am-11.30am
  - Content delivery
  - Pop-up quiz
  - In-class , Programming assignments

References

- Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2022. Introduction to algorithms. MIT Press.
- Neapolitan, R. and Naimipour, K., 2015. Foundations of Algorithms. 5th Edition, Jones & Bartlett Publishers.
- Karumanchi, N., 2017, Data Structures and Algorithms Made Easy

Course Structure

- Tutorial / Practical sessions [no swaps]
  - Monday 3.00pm – 5.00pm Group 1
  - Tuesday 3.00pm – 5.00pm Group 2
  - Tutorial delivery
  - Discussion of problems
  - Quiz
  - Take home programming assignment

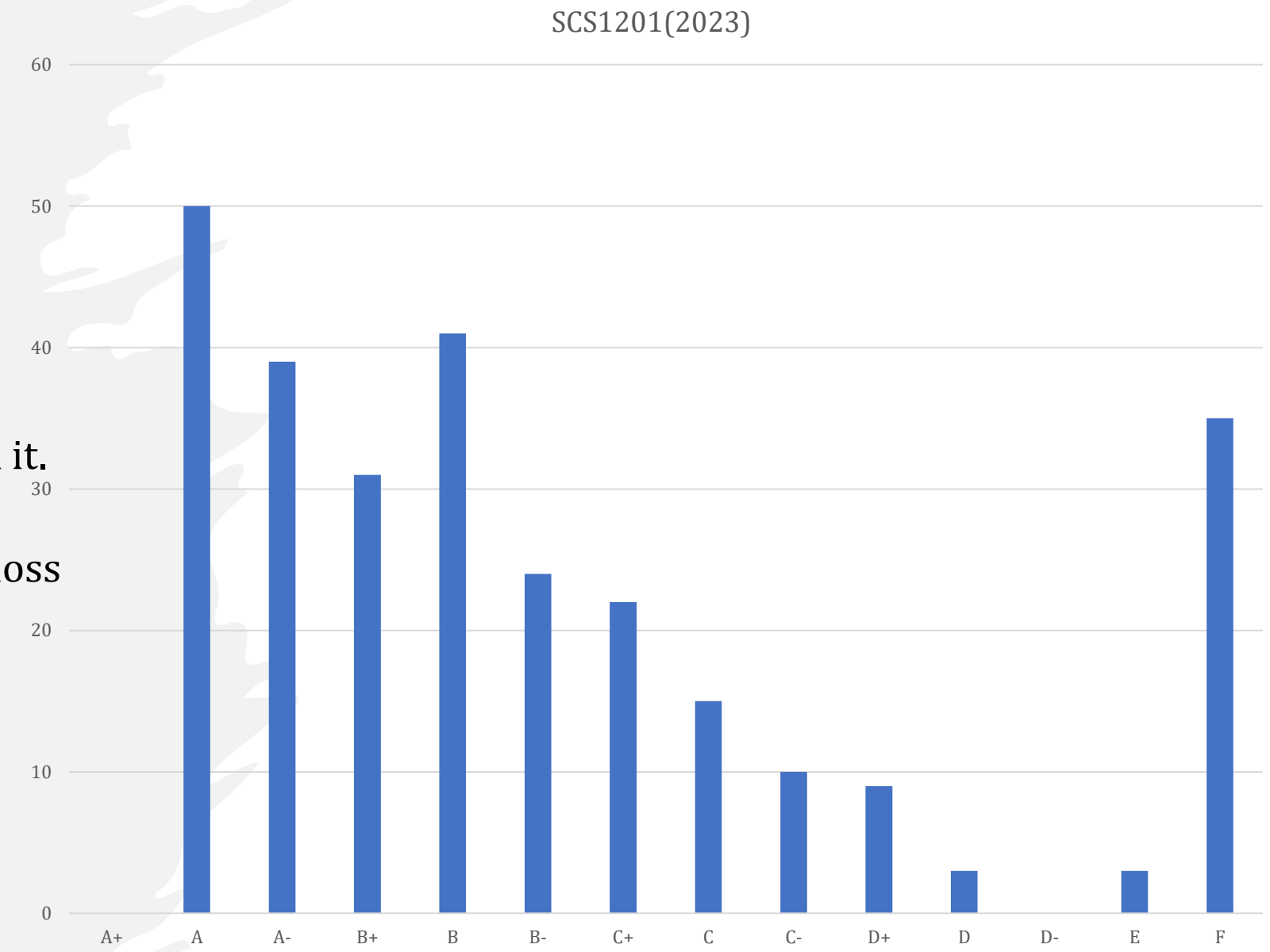
Expectations

- 4 credits (3L + 1P)

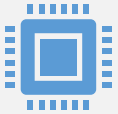
- Notional hours = 3 x 50 +100 = 250 hrs
- Lecture hours = 3 x 15 week = 45 hrs
- Practical hours = 30 hrs
- 1 L hr Self study 2 hrs
- Self study hours = 6 x 15 = 90hrs (per semester) , 3 hrs per week
- Remaining course work = 250-45-30 -90 = 85 hrs = 5.6 hrs per week

Makeup assignment / Plagiarism

- No makeup assignments
- If you missed it , you missed it.
- Takehome/ Programming assignments – check with moss tool.



Problem Instances



An *instance* is the actual data for which the problem needs to be solved.

We use the terms *instance* and *input* interchangeably.

Problem: Sort list of records.
Instances:

(1, 10, 5)

(1, 2, 3, 4, 1000, 27)

Time complexity analysis is done in terms of input size

Efficiency

The efficiency of an algorithm depends on the quantity of resources it requires

Usually we compare algorithms based on their *time*

Sometimes also based on the *space* they need.

The time required by an algorithm depends on the instance *size* and its *data*

Example: Sequential search

Problem: *Find a search key in a list of records*

Algorithm: Sequential search

Main idea: Compare search key to all keys until a match is found or list is exhausted

Time depends on the size of the list *n* and the data stored in a list

What is an algorithm ?

- An algorithm is the step-by-step unambiguous instructions to solve a given problem.

- Let us consider the problem of preparing an omelette. To prepare an omelette, we follow the steps given below:

```
1) Get the frying pan.
2) Get the oil.
    a. Do we have oil?
            i. If yes, put it in the pan.
            ii. If no, do we want to buy oil?
                    1. If yes, then go out and buy.
                    2. If no, we can terminate.
3) Turn on the stove, etc...
```

What is an algorithm ?

An algorithm is the step-by-step unambiguous instructions to solve a given problem.

How to judge merits of algorithm ?

- Correctness – does the algorithm give solution to the problem in a finite number of steps ?
- Efficiency – how much resources (memory / time) does it take to execute ?

Why the Analysis of Algorithms?

- Multiple algorithms are available for solving the same problem (eg: sorting problems – many algorithms)
- Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.

Goal of the analysis of algorithms

Compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developer effort, etc.)

What is Running Time Analysis?

- Process of determining how processing time increases as the size of the problem (input size) increases.
- Input size is the number of elements in the input, and depending on the problem type, the input may be of different types.
- Common types of inputs.
  - Size of an array
  - Polynomial degree
  - Number of elements in a matrix
  - Number of bits in the binary representation of the input
  - Vertices and edges in a graph

How to compare algorithms ?

Execution times?

- Execution times are specific to a particular computer.

Number of statements executed?

- Number of statements varies with the programming language as well as the style of the individual programmer.

Ideal solution?

- Express the running time of a given algorithm as a function of the input size n (i.e., f(n)) and compare these different functions corresponding to running times.
- This comparison is independent of machine time, programming style, etc.

What is Rate of Growth ?

- Running time increases as a function of input.
- Eg: Assume that you go to a shop to buy a car and a bicycle. If your friend sees you there and asks what you are buying, then in general you say buying a car.
- This is because the cost of the car is high compared to the cost of the bicycle (approximating the cost of the bicycle to the cost of the car).

$$Total\ Cost = cost\ of\ car + cost\ of\ biycycle$$

Total Cost $\sim$ cost of car (approximation)

What is Rate of Growth ? [Cont'd]

$$n^4 + 2n^2 + 100n + 500 \sim n^4$$

Asymptotic Notation

- A way to describe the running time or space complexity of an algorithm based on the input size.
  - Commonly used in complexity analysis to describe how an algorithm performs as the size of the input grows.
  - Commonly used notations : Big O, Omega and Theta.
  - Choice of asymptotic notation depends on the problem and the specific algorithm used to solve it.

Time Complexity Analysis

Best Case: The smallest amount of time needed to run any instance of a given size

Worst Case: The largest amount of time needed to run any instance of a given size

Average Case: the expected time required by an instance of a given size

Time Complexity Analysis

- If the *best*, *worst* and *average* “times” of some algorithms are identical, we have ***every case time analysis*.**

  e.g., array addition, matrix multiplication, etc.

- Usually, the best, worst and average time of an algorithm are different.

Worst case time analysis

- Drawbacks of comparing algorithms based on their worst case time:

  - An algorithm could be is not superior. superior on average than another, although the worst case time complexity

  - For some algorithms a worst case instance is very unlikely to occur in practice.

Evaluation of runtime through experiments

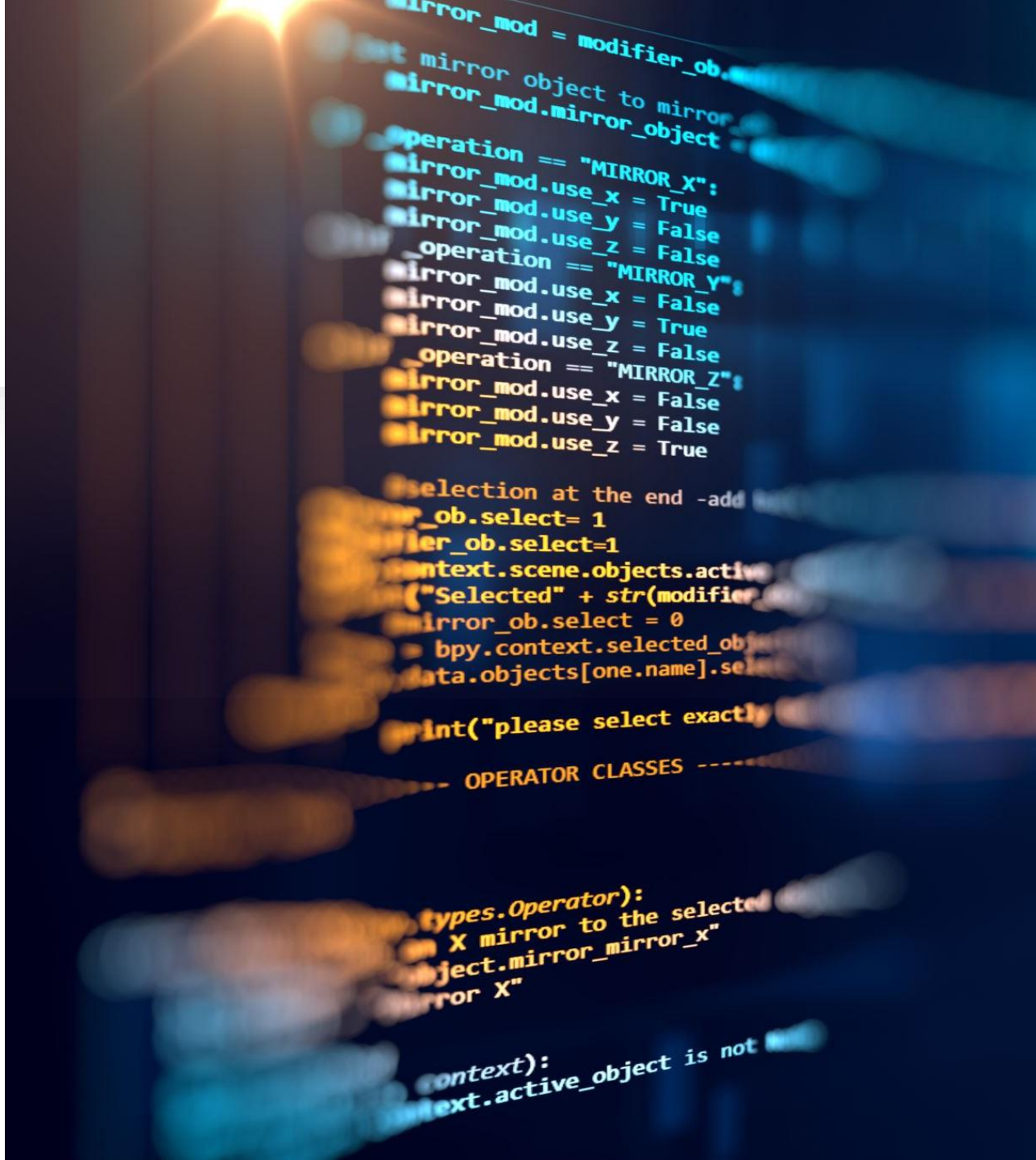
- Challenges
  - Algorithm must be fully *implemented*
  - To compare runtime we need to use the *same hardware* and *software* environments
  - Different *coding style* of different individuals'
- Is there any better way?

Requirements for time complexity analysis

- Independence
- A priori
- Large instances
- Growth rate classes

Independence Requirement

- Time complexity analysis must be *independent* of:
  - *The hardware of a computer*
  - *The programming language used for pseudo code*
  - *The programmer that wrote the code*



A Priori Requirement

- Analysis should be a priori; that is, it should be done *before* implementing the algorithm

- Derived for any algorithm expressed in high level description or pseudo code

Large Instance Requirement

- Algorithms running efficiently on small instances may run very slowly with large instance sizes

- Analysis must capture algorithm behavior when problem instances are large
  - For example, linear search may not be efficient when the list size n = 1,000,000

Growth Rate Classes Requirement

- Time complexity analysis must classify algorithms into:
  - Ordered classes so that all algorithms in a single class are considered to have the same efficiency.
  - If class A “is better than“ class B, then all algorithms that belong to A are considered more efficient than all algorithms in class B.

Growth rate classes

- Growth rate classes are derived from instruction counts
- Time analysis partitions algorithms into general equivalence classes such as:
  - logarithmic,
  - linear,
  - quadratic,
  - cubic,
  - polynomial
  - exponential, etc.

Instruction counts

- Provide rough estimates of actual number of instructions executed
- Depend on:
  - Language used to describe algorithm
  - Programmer's style
  - Method used to derive count
- Could be quite different from actual counts
- Algorithm with count=2n, may not be faster than one with count=5n.

Comparing an nlogn to an n^2 algorithm

- An nlogn algorithm is always more efficient for *large* instances.

- Pete is a programmer for a super computer. The computer executes 100 million instructions per second. His implementation of Insertion Sort requires $2n^2$ computer instructions to sort n numbers.

- Joe has a PC which executes 1 million instructions per second. Joe's sloppy implementation of Merge Sort requires $75n\lg n$ computer instructions to sort n numbers.

Who sorts a million numbers faster?

Super Pete:

$(2 (10^6)^2$ instructions) / (10^8 instructions/sec)
= 20,000 seconds
≈ 5.56 hours

Average Joe:

(75 *10^6 lg (10^6) instructions)/ (10^6 instructions/sec)
= 1494.8 seconds **≈ 25 minutes**

Who sorts 50 numbers faster?

Super Pete:

(2 $(50)^2$ instructions) / (10^8 instructions/sec)
≈ 0.00005 seconds

Average Joe:

(75 *50 lg (50) instructions) / (10^6 instructions/sec)
≈ 0.000353 seconds

Computing Instruction Counts

- Given a (non-recursive) algorithm expressed in pseudo code we explain how to:
  - Assign counts to high level statements
  - Describe methods for deriving an instruction count
  - Compute counts for several examples

Assignment Statement

1. A= B*C-D/F

- $Count_1 = 1$
- In reality? At least 4

Note: When numbers B, C, D, F are very large (a number can't be stored in a single word), algorithms that deal with large numbers will be used and the count will depend on the number of digits needed to store the large numbers.

Loop condition

1. (i < n) && (!found)

- $Count_1 = 1$

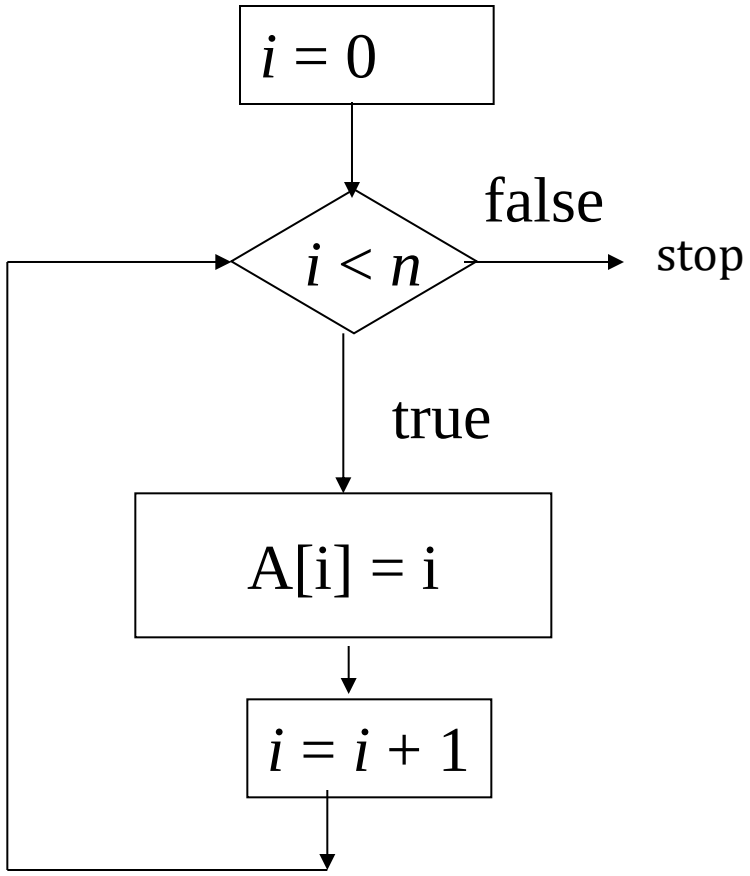
Note: if loop condition invokes a function, count of the function must be used

for loop body

1. for (i=0; i < n; i++)
2. A[i] = i

$\text{Count}_2 = 1$

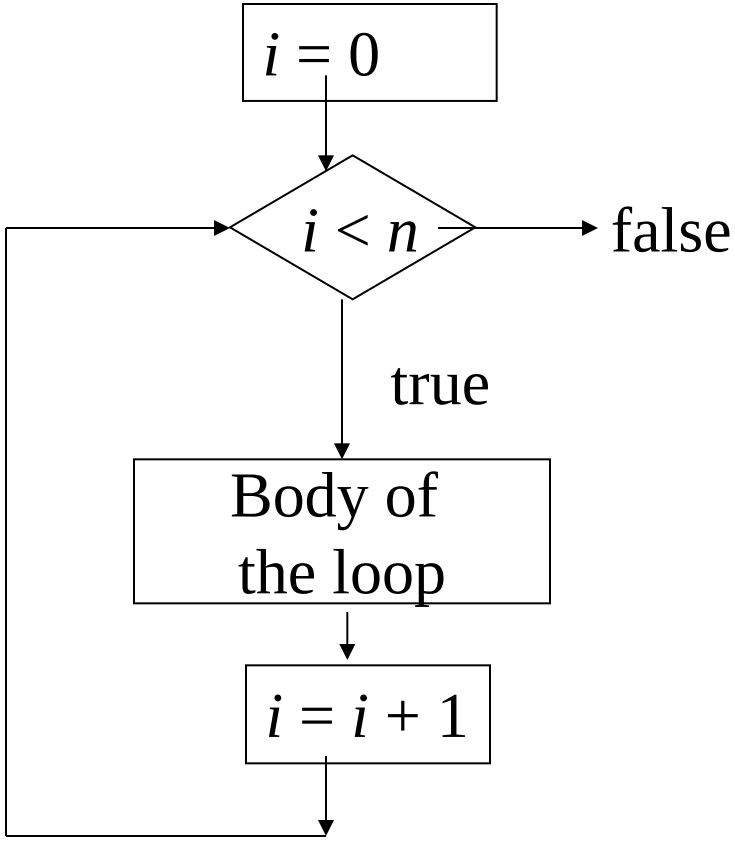
$\text{Count}_{1(2)} = \sum_{i=0}^{n-1} \text{Count}_2$



for loop control

1. for (i = 0; i < n; i++)
2. <body>

Count = number of times loop condition is executed (assuming loop condition has a count of 1)



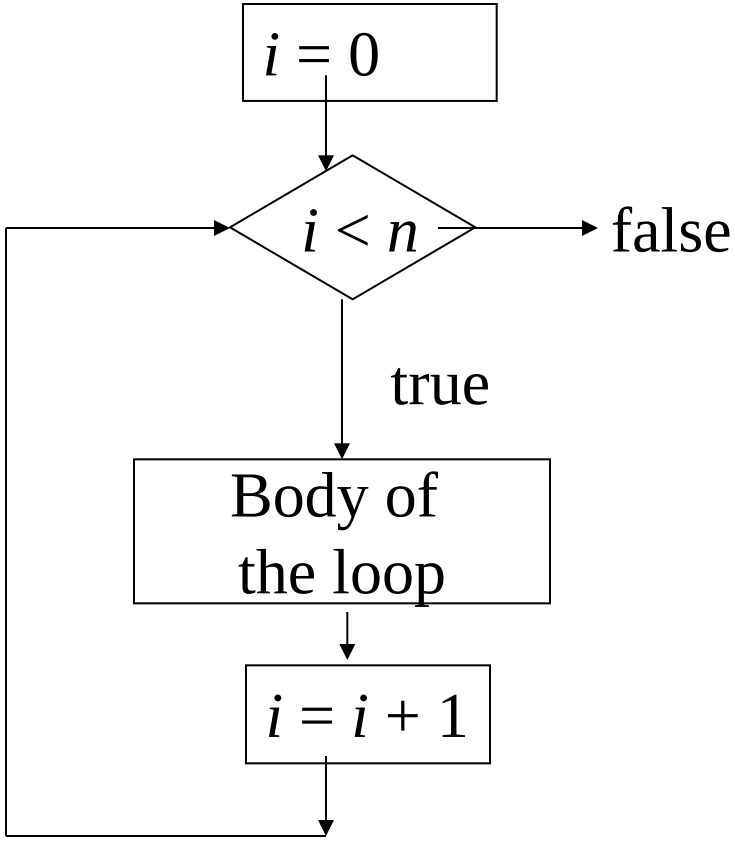
for loop control

1. for (i=0; i < n; i++)
2. <body>

$Count_1$ = number times loop condition i < n is executed

$= n + 1$

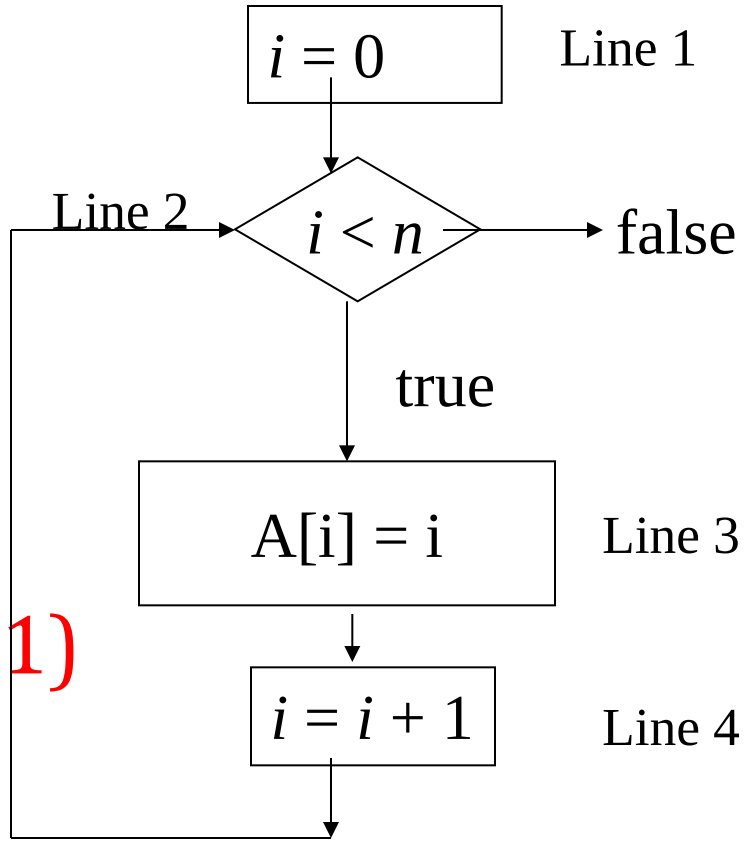
Note: last time condition is checked when i =n and (i < n) evaluates to false



while loop control

```
1. i = 0
2. while (i < n){
3.     A[i] = i
4.     i = i + 1}
```

Count = number of times loop condition is executed (assuming loop condition has a count of 1)

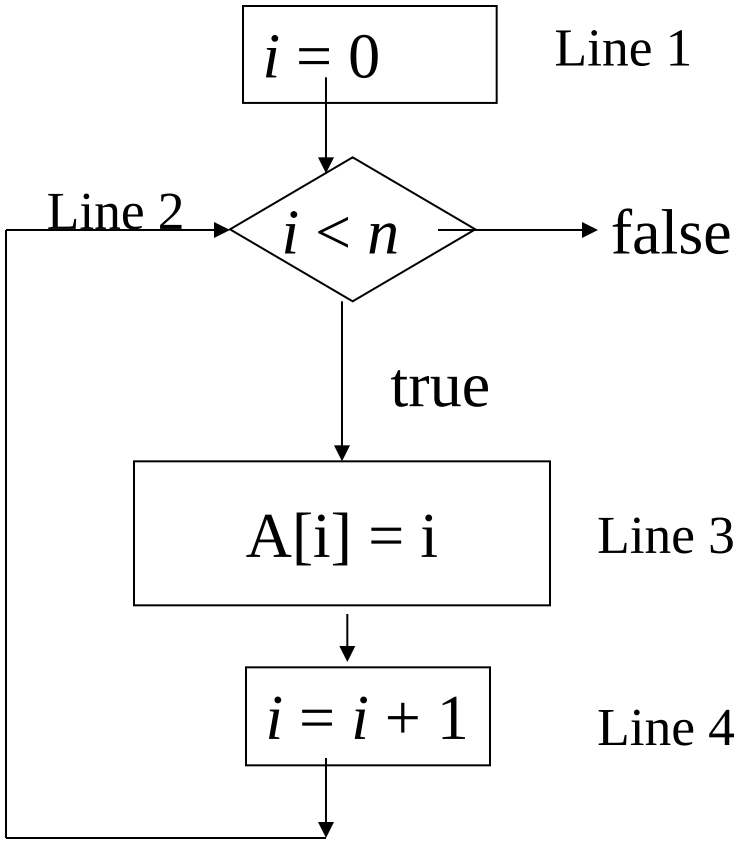


while loop control

```
1. i = 0
2. while (i < n){
3.     A[i] = i
4.     i = i + 1}
```

$Count_2$ = number of times loop condition (i < n) is executed

= n + 1



If statement

```
Line 1: if (i == 0)
Line 2:   statement
        else
Line 3:   statement
```

For worst case analysis, how many counts are there for Count_{if}?

$$\text{Count}_{\text{if}} = 1 + \max\{\text{count2}, \text{count3}\}$$

Asymptotic Analysis Guidelines

1. Loops: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
//execute n times
for(i=1; i<= n ; i++)
        m =  m+2; //constant time c
```

Total time = c x n = cn = O(n)

Asymptotic Analysis Guidelines

2. Nested loops : Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
//execute outer loop n times
for(i=1; i<= n ; i++){
        //inner loop executes n time
        for(j=1; j<= n ; j++){
                k =  k+1; //constant time
}}
```

Total time = c x n x n = $cn^2 = O(n^2)$

Asymptotic Analysis Guidelines

3. Consecutive statements : add the time complexities of each statement.

```
x= x +1; //constant time
//execute n times
for(i=1; i<= n ; i++){
        m = m +2;} //constant time
        //outer loop executes n time
        for(i=1; i<= n ; i++){
                //inner loop executed n times
                for(j=1; j<= n ; j++){
                        k =  k+1; //constant time
}}
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$

Asymptotic Analysis Guidelines

4. if-then-else-statement : Worst-case running time: the if, plus either the then part or the else part (whichever is the larger).

```
//constant
if(length() == 0) {
        return false;  //constant
}
else{  //(constant + constant )*n
        for(int n=0; n< length; n++){
                if(list[n].equals(otherList.list[n]) //constant{
                        return false;
                }
}
```

Total time = $c_0 + c_1 + (c_2 + c_3)*n = O(n)$

Asymptotic Analysis Guidelines

5. Logarithmic Complexity : An algorithm is O(logn) if it takes a constant time to cut the problem size by a fraction (usually by ½).

```
for(i=1; i<= n; i=i*2)
```

Taking logarithm on both side
$\log(2^k) = \log n$
klog 2= logn
k = logn // if we assume base 2

Total time = O(logn)

How about the case below ?

```
for(i=n; i >= 1; i=i/2)
```

Total time = O(logn)

Example 1: Method 1

```
1.   for (i=0;  i<n;  i++ )
2.       A[i] = i
```

- Method 1

$count_1 = n+1$

$count_{1(2)} = n*1 = n$

Total = $(n+1)+n = 2n+1$

Example 1: Method 2

- Method 2

```
1.   for (i=0;  i<n;  i++ )
2.       A[i] = i + 1
```

- Barometer operation = + in body of loop

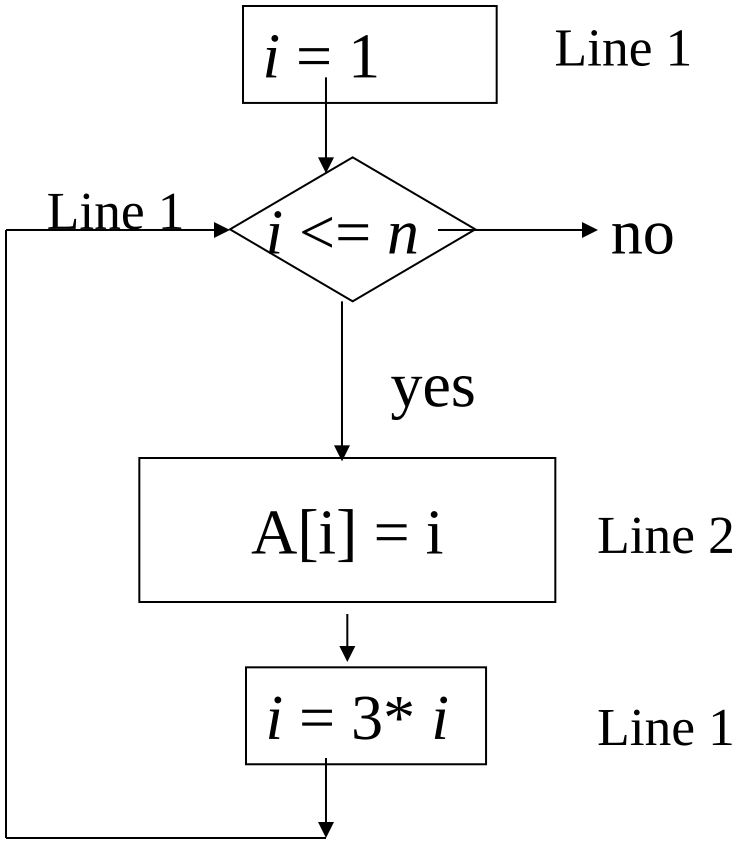
- $count_{1(+)} = n$

Barometer Method

- Used to analyze time complexity of recursive/ iterative algorithms.
- Involves identifying a representative “barometer operation”
  - Fundamental operation that dominate cost of the algorithm
  - Then derive a recurrence relation
- Solving this recurrence gives the overall time complexity of the algorithm.
- Choice of barometer operation – ensure operation is a good proxy for measuring the algorithm’s cost.
  - Sorting algorithm – comparisons
  - Matrix multiplication – multiplication
  - Graph traversal – edge relaxation

Example 2: What is $count_{1(2)}$?

```
1.for (i=1;i<=n;i=3*i)
2.    A[i] = i
```



Example 2: What is $count_{1(2)}$?

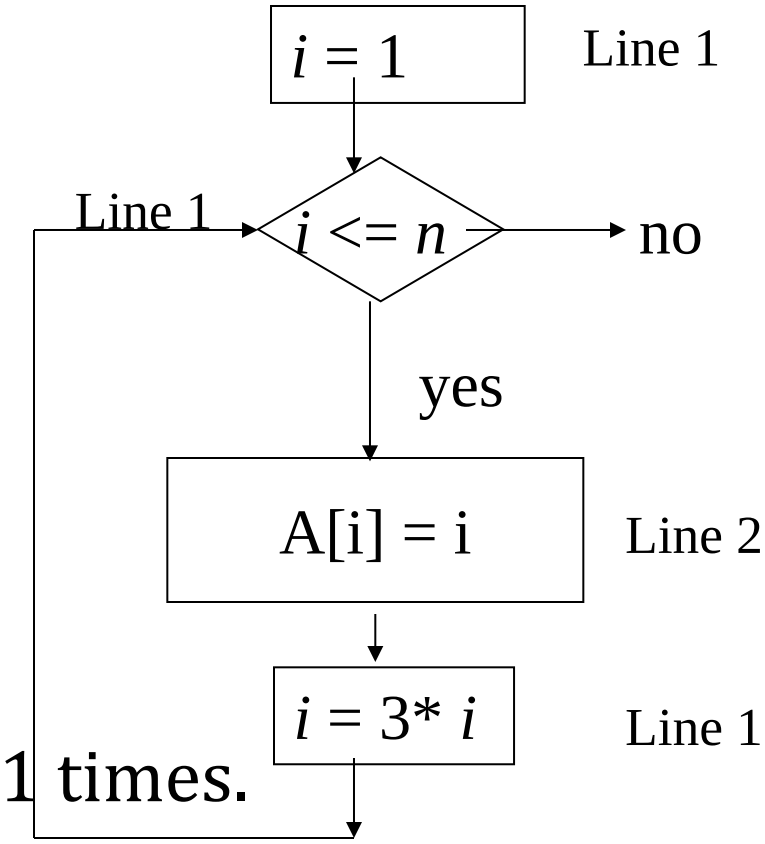
```
1. for(i=1;i<=n; i=3*i)
2.    A[i] = i
```

For simplicity, $n = 3^k$ for some positive integer k.

Body of the loop executed for i = $1(=3^0), 3^1, 3^2,\ldots,3^k$.

So $count_{1(2)} = \sum_{q=0}^{k} count_2 = k+1$

Since $k = \log_3 n$, it is executed $\log_3 n + 1$ times.



Example 3: Sequential Search

```
1.  location=0
2.  while (location<=n-1
3.        && L[location]! = x)
4.     location++
5.  return location
```

- Barometer operation = (L[location]! = x?)
- Best case analysis

  x == L[0] and the count is 1

- Worst case analysis

  x = L[n-1] or x not in the list. Count is *n*.

Example 4:

1. x = 0
2. for (i=0; i<n; i++)
3. for (j=0, j<m; j++)
4. x = x + 1

Barometer is + in body of loop.

$count_{2(3(+))}$ = ?

$$\sum_{i=0}^{n-1} count_{3(+)} = \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} count_{+} =$$

$$= \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} 1 = \sum_{i=0}^{n-1} m = m\sum_{i=0}^{n-1} 1 = mn$$

Example 5:

1. x=0
2. **for** (i=0; i<n; i++)
3. **for** (j=0, j<n^2; j++)
4. x = x + 1

- $Count_{2(3(+))}$= ?

Answer: $n*n^2*1$

Example 6:

```
Line 1: for (i=0; i<n; i++)
Line 2:     for (j=0, j<i; j++)
Line 3.         x = x + 1
```

Barometer operator = +

$\text{Count}_{1(2(+))}$= ? $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}1=\sum_{i=0}^{n-1}i=(n-1)n/2$

Example 7:

```
1. for (i=0; i<n; i++)
2.    for (j=0, j<i; j++)
3.       for (k=0; k<=j; k++)
4.             x++;
```

$\text{count}_{1(2(3))} = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} \sum_{k=0}^{j} 1 =$

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

$$\frac{1}{6}(n-1)n(n+1)$$

Note: $1^2 + 2^2 + \ldots + n^2 = \frac{1}{6} n(n+1)(2n+1)$

Asymptotic Growth Rate

Asymptotic Notation

- Asymptotic notation does not provide an exact running time or space usage for an algorithm, but rather a description of how the algorithm scales with respect to input size.
- A useful tool for comparing the efficiency of different algorithms and for predicting how they will perform on large input sizes.

Big – O (Big-Oh)

- Upper bound on the growth rate of an algorithm's running time or space usage.
- Represent worst case scenario - the maximum amount of time or space an algorithm may need to solve a problem.
- Eg: What does it mean to say O(n) ?
  - It means that the running time of the algorithm increases linearly with the input size n or less.

Are they true? Why or why not?

- **1,000,000 n^2 $\in O(n^2)$?**
- **True**

- **$(n - 1)n / 2 \in O(n^2)$?**
- **True**

- **$n / 2 \in O(n^2)$?**
- **True**

- **$\lg(n^2) \in O(\lg n)$?**
- **True**

- **$n^2 \in O(n)$?**
- **False**

Omega notation (Ω)

- Provides a lower bound on the growth rate of an algorithm's running time or space usage.
- Represent best case scenario – minimum amount of time or space an algorithm may need to solve a problem.
- Eg: What does it mean to say Ω(n) ?
  - It means that the running time of the algorithm increases linearly with the input size n or more.

Are they true?

- **1,000,000 $n^2 \in \Omega(n^2)$ why /why not?**
  - **true**
- **$(n - 1)n / 2 \in \Omega(n^2)$ why /why not?**
  - **true**

- **$n / 2 \in \Omega(n^2)$ why /why not?**
  - **(false)**
- **$\lg(n^2) \in \Omega(\lg n)$ why /why not?**
  - **(true)**
- **$n^2 \in \Omega(n)$ why /why not?**
  - **(true)**

Theta notation (Θ)

- Provides both an upper and lower bound on the growth rate of an algorithm's running time or space usage.
- Represents the average-case scenario, i.e., the amount of time or space an algorithm typically needs to solve a problem
- Eg: What does it mean to say Θ(n) ?

Then it means that the running time of the algorithm increases linearly with the input size n.

More Θ

- **1,000,000 $n^2 \in \Theta(n^2)$ why /why not?**
  - **True**
- **$(n - 1)n / 2 \in \Theta(n^2)$ why /why not?**
  - **True**
- **$n / 2 \in \Theta(n^2)$ why /why not?**
  - **False**
- **$\lg(n^2) \in \Theta(\lg n)$ why /why not?**
  - **True**
- **$n^2 \in \Theta(n)$ why /why not?**
  - **False**

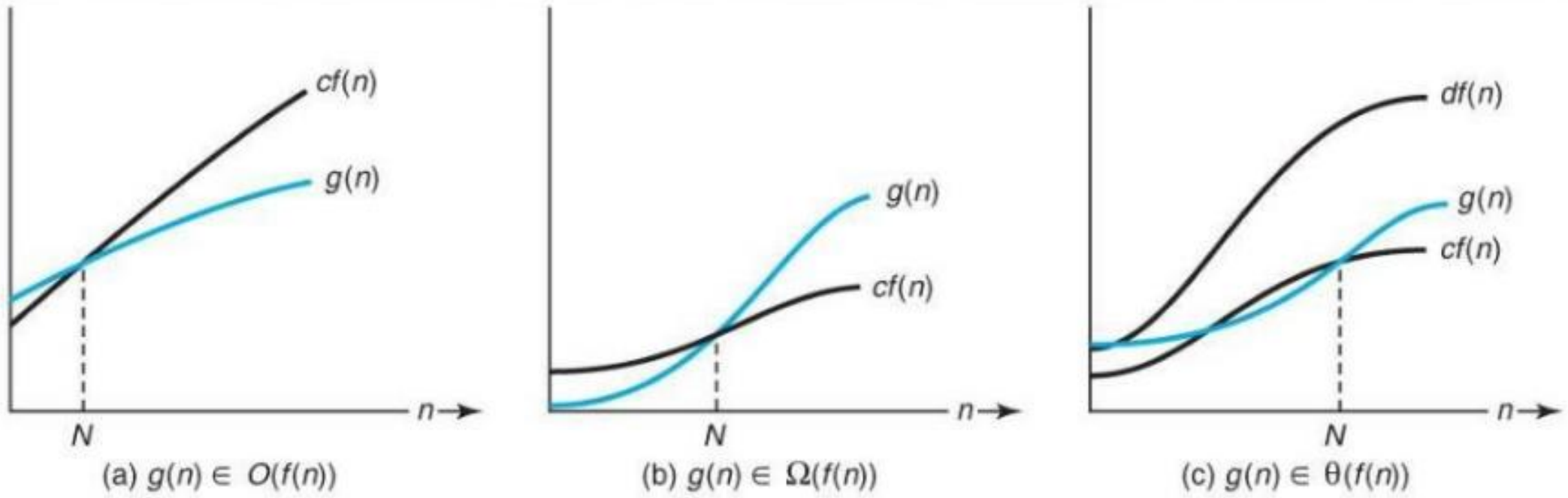


Illustration : Big O, Omega and Theta

small o

- $o(f(n))$ is the set of functions $g(n)$ which satisfy the following condition:

  For *every* positive real constant c, there exists a positive integer N, for which

  $$g(n) \leq cf(n) \text{ for all } n \geq N$$

Big O vs. Small o

Big O

- f(n)=O(g(n)) means f(n) grows at most as fast as g(n)

$$f(n) \leq C \cdot g(n) \quad \text{for all } n \geq n_0,$$

- Big O gives an upper bound on f(n)
- f(n) grow as fast as g(n) but not faster.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \leq C, \quad C > 0.$$

Small o

- f(n) = o(g(n) means f(n) grows strictly slower than g(n)

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

- Small o gives a strict upper bound
- f(n) is negligible compared to g(n) as n→ infi.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

small omega

- $\omega(f(n))$ is the set of functions $g(n)$ which satisfy the following condition:

  For *every* positive real constant c, there exists a positive integer N, for which

  $g(n) \geq cf(n)$ for all $n \geq N$

Omega Ω vs. Small ω

Omega Ω

- f(n)=Ω (g(n)) means f(n) grows at least as fast as g(n)

$$f(n) \geq C \cdot g(n) \quad \text{for all } n \geq n_0,$$

- Omega gives an lower bound on f(n)
- f(n) grow as fast or at the same rate as g(n) but not slower.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \leq C,$$
$$C > 0.$$

Small ω

- f(n) = ω(g(n) means f(n) grows strictly faster than g(n)

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

- Small ω gives a strict lower bound
- f(n) must grow strictly faster than g(n), g(n) becomes negligible compared to g(n) as n→ ∞.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Analogy between asymptotic comparison of functions and comparison of real numbers.

$$f(n) = O(g(n)) \quad \approx \quad a \leq b$$
$$f(n) = \Omega(g(n)) \quad \approx \quad a \geq b$$
$$f(n) = \Theta(g(n)) \quad \approx \quad a = b$$
$$f(n) = o(g(n)) \quad \approx \quad a < b$$
$$f(n) = \omega(g(n)) \quad \approx \quad a > b$$

f(n) is asymptotically smaller than g(n) if $f(n) = o(g(n))$

f(n) is asymptotically larger than g(n) if $f(n) = \omega(g(n))$

Thank you