# Foundations of Algorithm SCS1308

Dr. Dinuni Fernando PhD

Senior Lecturer

## 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-11.00am
  - Content delivery
  - Pop-up quiz
- 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

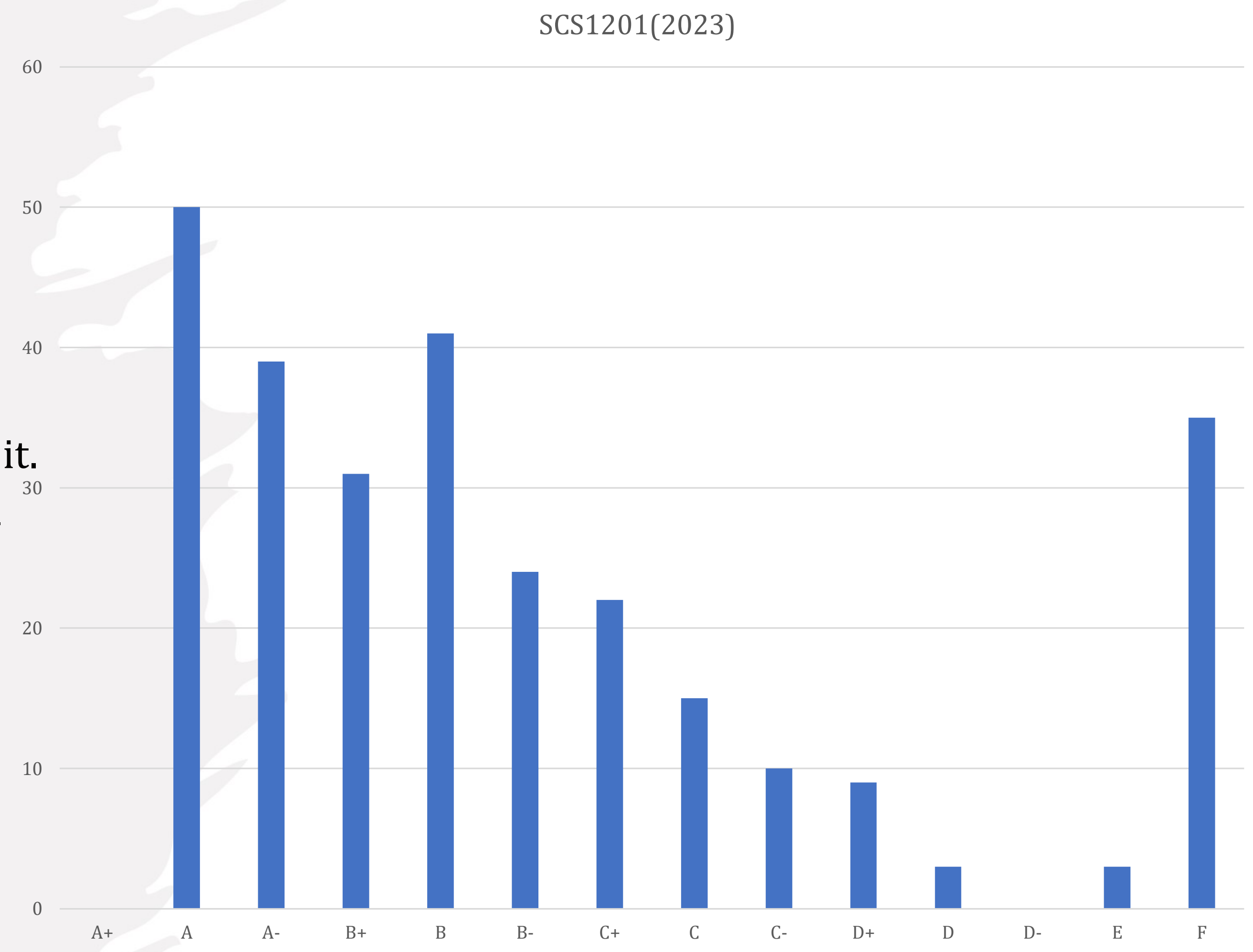
## Expectations

- 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.
- 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

## Reminder: Instance Size

Formally, *size = number of bits needed to represent the instance* in a computer

We will usually be much less formal

For example, we assume that the size of a record is a constant number of bits *c* for sorting → Consider that the input size is n if n numbers to sort are given

## Size Examples

**Search and sort**

- Size = *n* number of records in the list. For search ignore search key

**Graphs problems**

- Size = (|V| + |E|)
- |V|: number of nodes
- |E|: number of edges

**Matrix problems**

- Size = r*c
- r: number of rows
- c: number of columns

## Exceptions: Number problems

Problems where input numbers can become increasingly large.

Examples:

- Recall Fibonacci number
- Factorial of 10 (10!), $10^6$, $10^{15}$
- Operations (e.g., add and multiplication) of large numbers where a number is expressed using several words
- For these problems we should use the formal definition

# Example: Factorial

Compute factorial of an *n* bit number *v*. Its value is

$v! = v * v\text{-}1 * \ldots * 3 * 2 * 1$

- O($v$) multiplications where $2^{n-1} \leq v < 2^n$
- Exponential time algorithm

Example:

- $n$ = 100 bits (first bit is 1)
- $2^{99} \leq v < 2^{100}$
- $v > 0.6 * 10^{30}$

## 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?**

- **E**xecution 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).

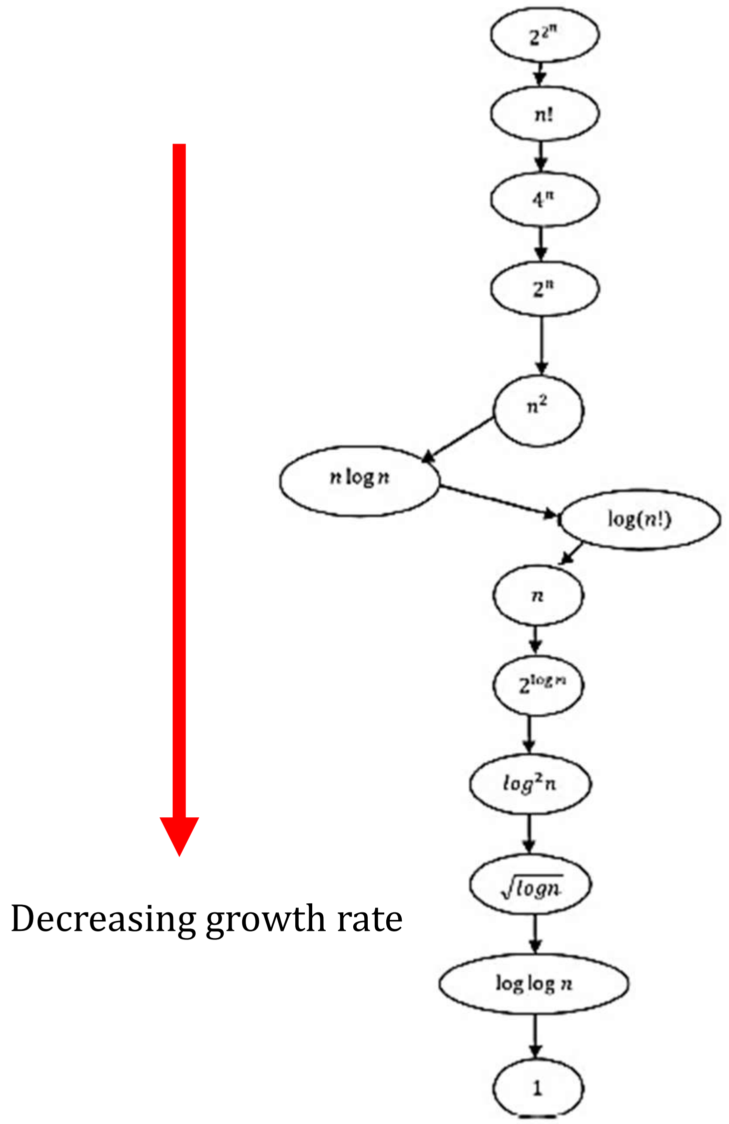
$$Total\ Cost = cost\ of\ car + cost\ of\ biycycle$$

Total Cost ~ cost of car (approximation)

## What is Rate of Growth ? [Cont’d]

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

# Commonly used growth rates



## Types of Analysis

- Worst case
  - Defines the input for which the algorithm takes a long time (slowest time to complete).
  - Input is the one for which the algorithm runs the slowest.
- Best case
  - Defines the input for which the algorithm takes the least time (fastest time to complete).
  - Input is the one for which the algorithm runs the fastest.
- Average case
  - Provides a prediction about the running time of the algorithm.
  - Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
  - Assumes that the input is random.

$$Lower\ bound \leq Average\ time \leq Upper\ bound$$

## 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.

## Time Analysis for Sequential search

- Worst-case: if the search key x is the last item in the array or if x is not in the array.

  $W(n) = n$

- Best-case: if x is matched with the first key in array S, which means x=S[1], regardless of array size n

  $B(n) = 1$

## Time Analysis for Sequential search

- Average-case: If the probability that x is in the kth array slot is 1/n:

$$A(n) = \sum_{k=1}^{n} (k \times \tfrac{1}{n}) = \tfrac{1}{n} \times \sum_{k=1}^{n} k = \tfrac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Question: What is A(n) if x could be outside the array?

## Worse case time analysis

- Most commonly used time complexity analysis

- *Because:*
    - Easier to compute than average case
    - Maximum time needed as a function of instance size
    - More useful than the best case

## 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'

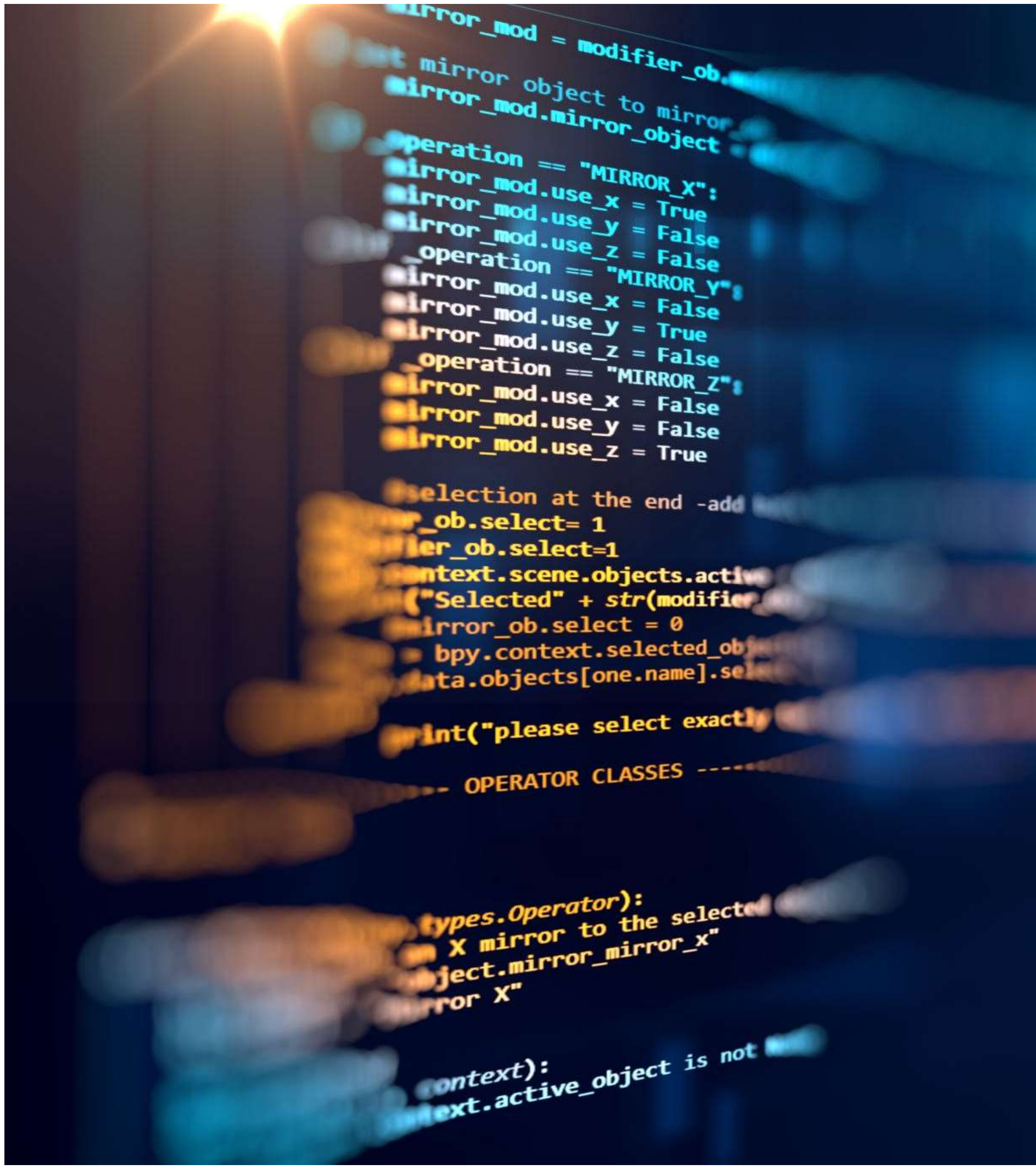
- Is there any better way?

## 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 <u>100 million instructions</u> per second. His implementation of Insertion Sort requires <u>$2n^2$</u> computer instructions to sort *n* numbers.
- Joe has a PC which executes <u>1 million instructions</u> per second. Joe's sloppy implementation of Merge Sort requires <u>$75n \lg n$</u> 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)
$\approx$ 0.00005 seconds

**Average Joe:**

(75 *50 lg (50 ) instructions) / ($10^6$ instructions/sec)
$\approx$ 0.000353 seconds

## Insertion sort

```
for i = 2 to n
  for (k = i; **k > 1 and a[k] < a[k-1]**; k--)
        swap (a[k], a[k-1])
```

→ ***invariant: a[1..i] is sorted***

Worst-case time complexity in terms of number of comparisons:

In the inner "for" loop, for a given i, the comparison is done at most i-1 times

In total:

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

$$W(n)=\frac{n(n-1)}{2}\in\Theta(n^2)$$

## Example: Binary search (Recursive)

```
Index Binsearch(index low, index high)
{
   index mid;

   if (low > high)  return 0;

  else
  {
     mid = floor[(low+high)/2];

     if (x == S[mid])  return mid;
     else if (x < S[mid])  return Binsearch(low, mid-1);
     else  return  Binsearch(mid+1, high);
  }
 }
```

Worst-case Time complexity:

$W(n) = W(n/2) + 1$
$W(1) = 1$

$\rightarrow$ $W(n) = \lg n + 1$

## *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

## Counts for High Level Statements

- Assignment
- loop condition
- for loop
    - for loop body
    - for loop control
- while loop
    - while loop control
    - while loop body
- if

Note: The counts we use are estimates; The goal is to derive a correct growth function

## *Assignment* Statement

1. A= B*C-D/F

- $\text{Count}_1 = 1$
- 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)

- $\text{Count}_1 = 1$

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

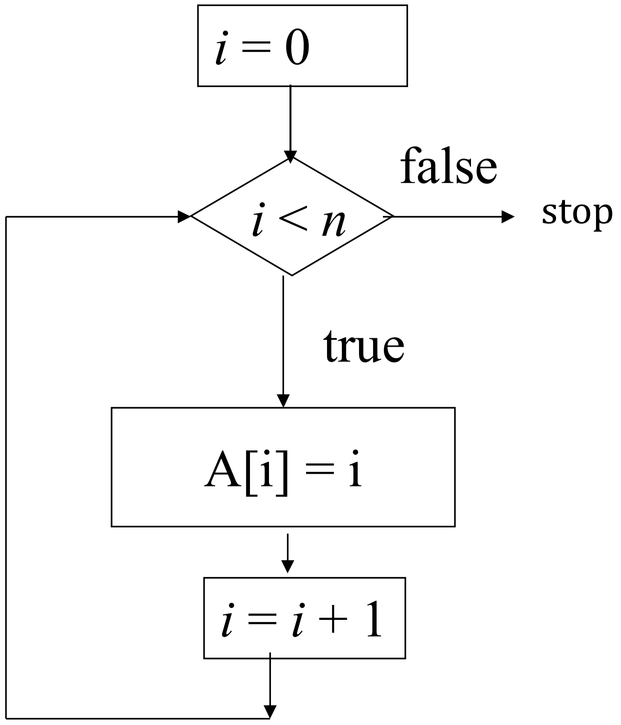
## for loop body

$\longrightarrow$

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

**Count$_2$ = 1**

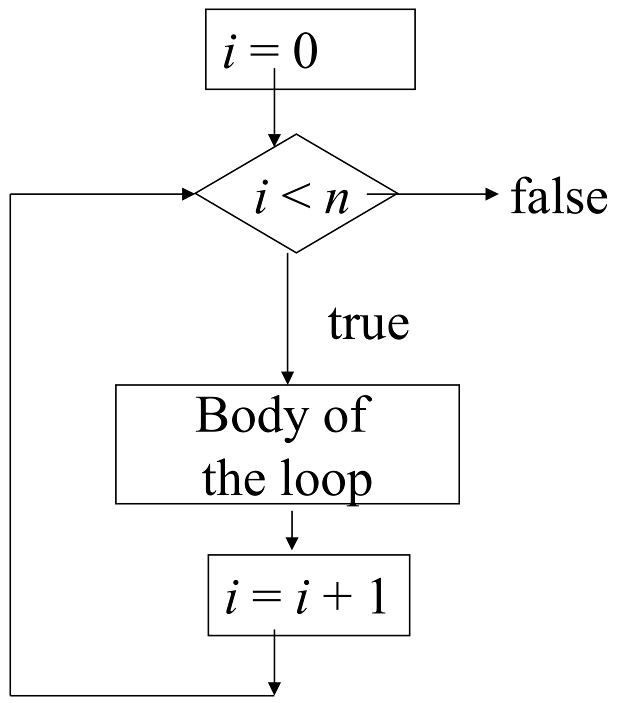
**Count$_{1(2)}$ =** $\sum_{i=0}^{n-1} \text{Count}_2$



## for loop control

$\longrightarrow$ 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)



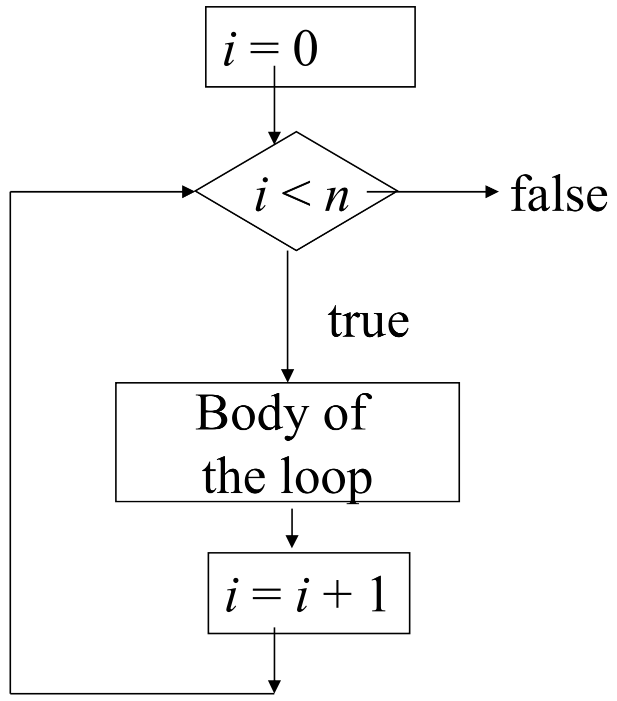
## for loop control

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

Count$_1$ = number times loop condition i < n is executed

$= n + 1$

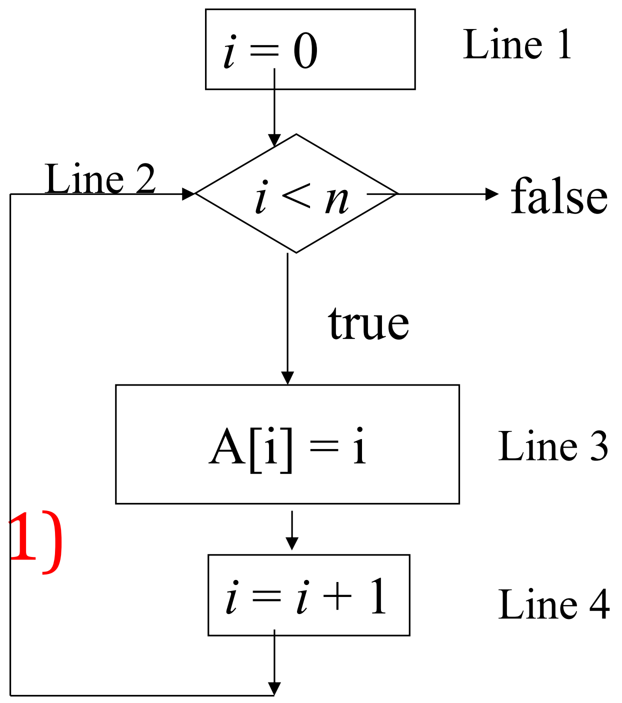
Note: last time condition is checked when i =n and (i < n) evaluates to false



## *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)

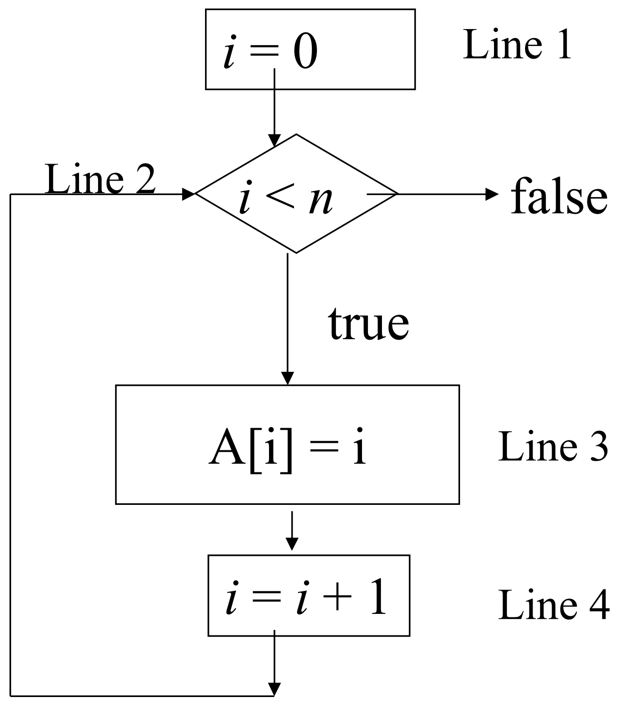


## *while* loop control

```
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 $\text{Count}_{if}$?

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

## 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 test, 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

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

$k\log 2 = \log n$

$k = \log n$ // 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

$count_1 = \quad n+1$

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

—————————————

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

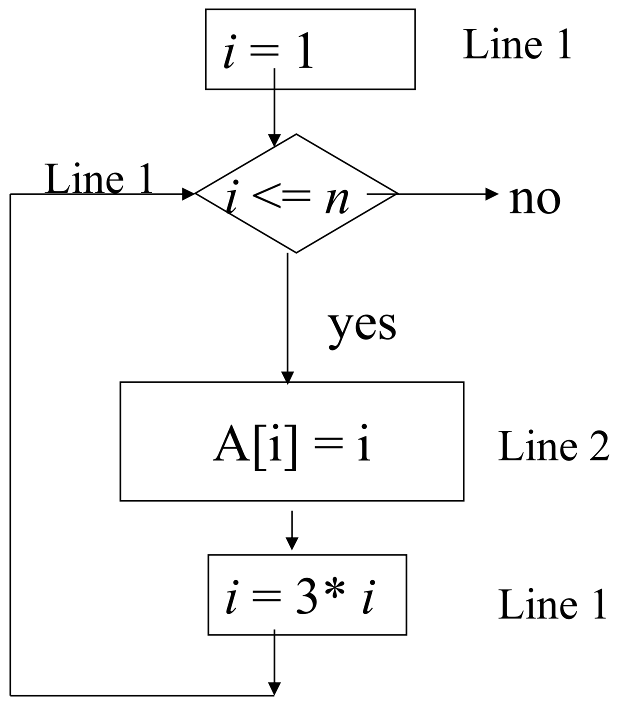
- $count_{1(+)} = n$

## Barometer Method

- Used to analyze time complexity of recursive algorithm.
- 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
```

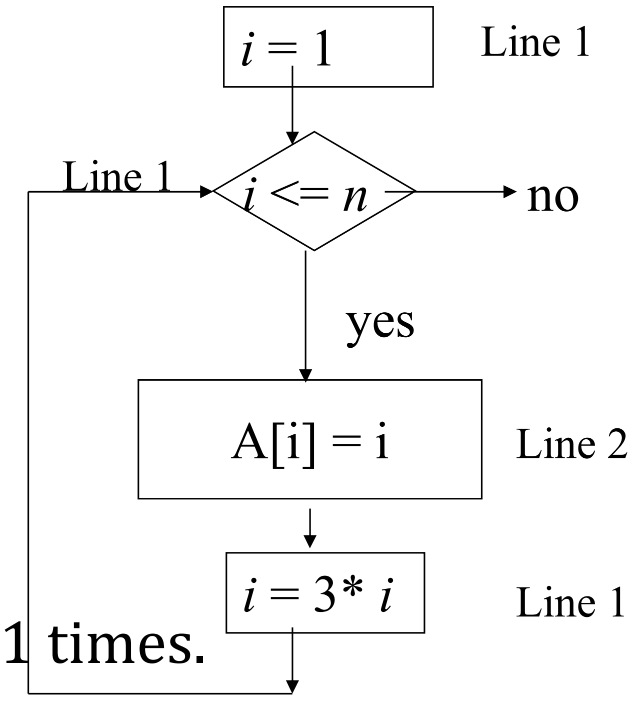
For simplicity, $n = 3^k$ for some positive integer k.

Body of the loop executed for

i = 1(=3$^0$), 3$^1$, 3$^2$,…,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

```
location=0
while (location<=n-1
      && L[location]! = x)
   location++
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
                 ↑
```

- $\text{Count}_{2(3(+))} = ?$

*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 = +

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

## 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++;
```

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

Thank you