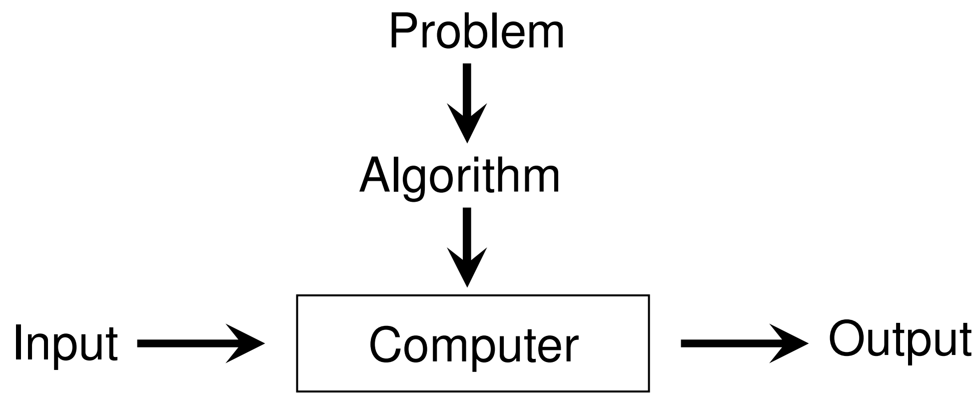
Module 1: Asymptotic Time Complexity and Intro to Abstract Data Types

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

What is an Algorithm?

- An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



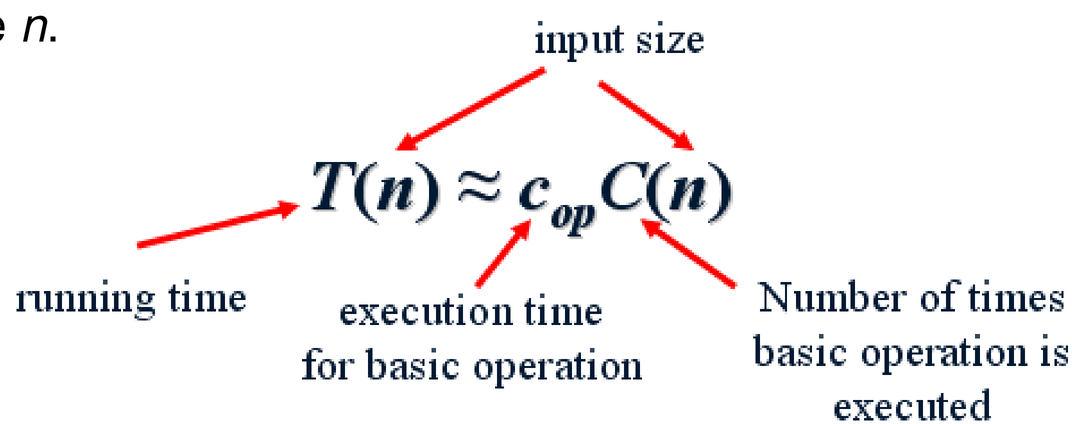
- Important Points about Algorithms
  - The non-ambiguity requirement for each step of an algorithm cannot be compromised
  - The range of inputs for which an algorithm works has to be specified carefully.
  - The same algorithm can be represented in several different ways
  - There may exist several algorithms for solving the same problem.
    - Can be based on very different ideas and can solve the problem with dramatically different speeds

The Analysis Framework

- **Time efficiency (time complexity):** indicates how fast an algorithm runs
- **Space efficiency (space complexity):** refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output
- Algorithms that have non-appreciable space complexity are said to be ***in-place***.
- The time efficiency of an algorithm is typically as a function of the input size (one or more input parameters)
  - Algorithms that input a collection of values:
    - The time efficiency of sorting a list of integers is represented in terms of the number of integers (*n*) in the list
    - For matrix multiplication, the input size is typically referred as n*n.
    - For graphs, the input size is the set of Vertices (V) and edges (E).
  - Algorithms that input only one value:
    - The time efficiency depends on the magnitude of the integer. In such cases, the algorithm efficiency is represented as the number of bits $1+\lfloor \log_2 n \rfloor$ needed to represent the integer n

Units for Measuring Running Time

- The running time of an algorithm is to be measured with a unit that is independent of the extraneous factors like the processor speed, quality of implementation, compiler and etc.
  - At the same time, it is not practical as well as not needed to count the number of times, each operation of an algorithm is performed.
- <u>Basic Operation:</u> The operation contributing the most to the total running time of an algorithm.
  - It is typically the most time consuming operation in the algorithm's innermost loop.
    - **Examples:** Key comparison operation; arithmetic operation (division being the most time-consuming, followed by multiplication)
  - We will count the number of times the algorithm's basic operation is executed on inputs of size *n*.



Examples to Illustrate Basic Operations

- <u>Sequential key search</u>
- Inputs: Array A[0…n-1], Search Key K
- Begin

```
        for (i = 0 to n-1) do
        if (A[i] == K) then
              return "Key K found at index i"
        end if
   end for
   return "Key K not found!!"
End
```

Best Case: 1 comparison
Worst Case: 'n' comparisons

- <u>Finding the Maximum Integer in an Array</u>
- Input: Array A[0…n-1]
- Begin

```
      Max = A[0]
        for (i = 1 to n-1) do
        if (Max < A[i]) then
              Max = A[i]
        end if
   end for
   return Max
End
```

Best Case: n-1 comparisons
Worst Case: n-1 comparisons

Note: Average Case number of Basic operations is the expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs

Why Time Complexity is important? Motivating Example

- An integer ‘n’ is prime if it is divisible (i.e., the remainder is 0) only by 1 and itself.
- Algorithm A (naïve)

```
Input n
Begin
  for i = 2 to n-1
    if (n mod i == 0)
      return "n is not prime"
    end if
  end for
  "return n is prime"
End
```

Best-case: 1 division
Worst-case: (n-1) – 2 + 1
= n-2 divisions
For larger n: ≈ n

Algorithm B (optimal)

```
Input n
Begin
  for i = 2 to √n
    if (n mod i == 0)
      return "n is not prime"
    end if
  end for
  "return n is prime"
End
```

Best-case: 1 division
Worst-case: √n – 2 +1
= √n – 1 divisions
For larger n: √n

Comparison of ‘n’ and ‘√n’

Input size (n)	Algorithm A (n)	Algorithm B(√n)
1	1	1
10	10	3.16
100	100	10
1000	1000	31.62
10000	10000	100
100000	100000	316.23
1000000	1000000	1000
10000000	10000000	3162.28

Orders of Growth

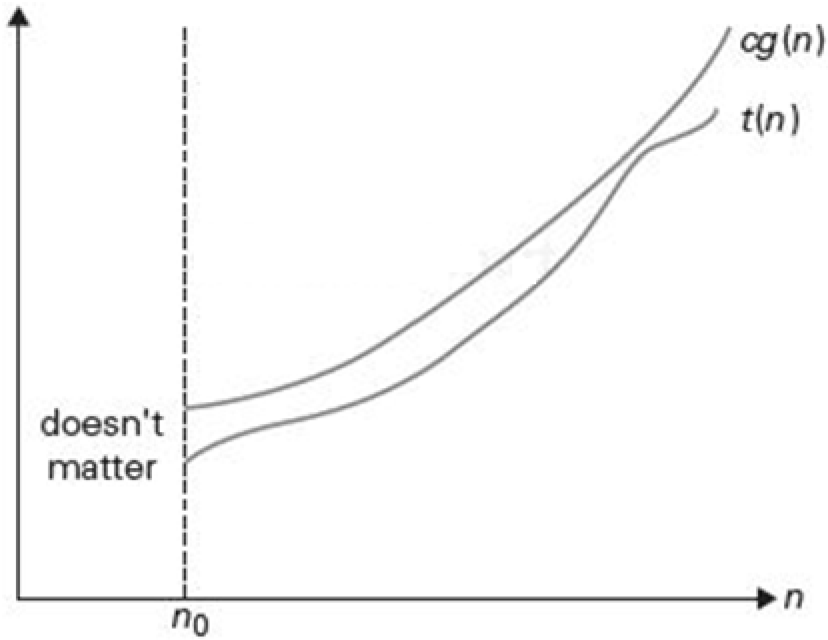
- We are more interested in the order of growth on the number of times the basic operation is executed on the input size of an algorithm.
- Because, for smaller inputs, it is difficult to distinguish efficient algorithms vs. inefficient ones.
- For example, if the number of basic operations of two algorithms to solve a particular problem are *n* and n^2 respectively, then
  - if $n = 3$, then we may say there is not much difference between requiring 3 basic operations and 9 basic operations and the two algorithms have about the same running time.
  - On the other hand, if $n = 10000$, then it does makes a difference whether the number of times the basic operation is executed is *n* or n^2.

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Exponential-growth functions

Source: Table 2.1 From Levitin, 3rd ed.

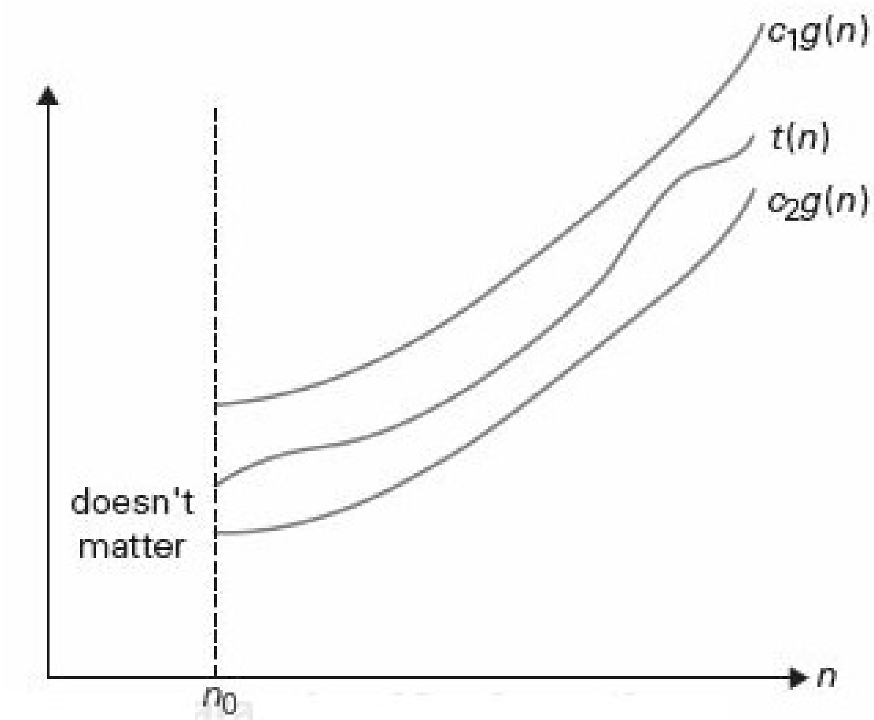
Asymptotic Notations: Formal Intro



$t(n) = O(g(n))$

$t(n) \le c*g(n)$ for all $n \ge n_0$

c is a positive constant (> 0)
and n_0 is a non-negative integer



$t(n) = \Theta(g(n))$

$c2*g(n) \le t(n) \le c1*g(n)$ for all $n \ge n_0$

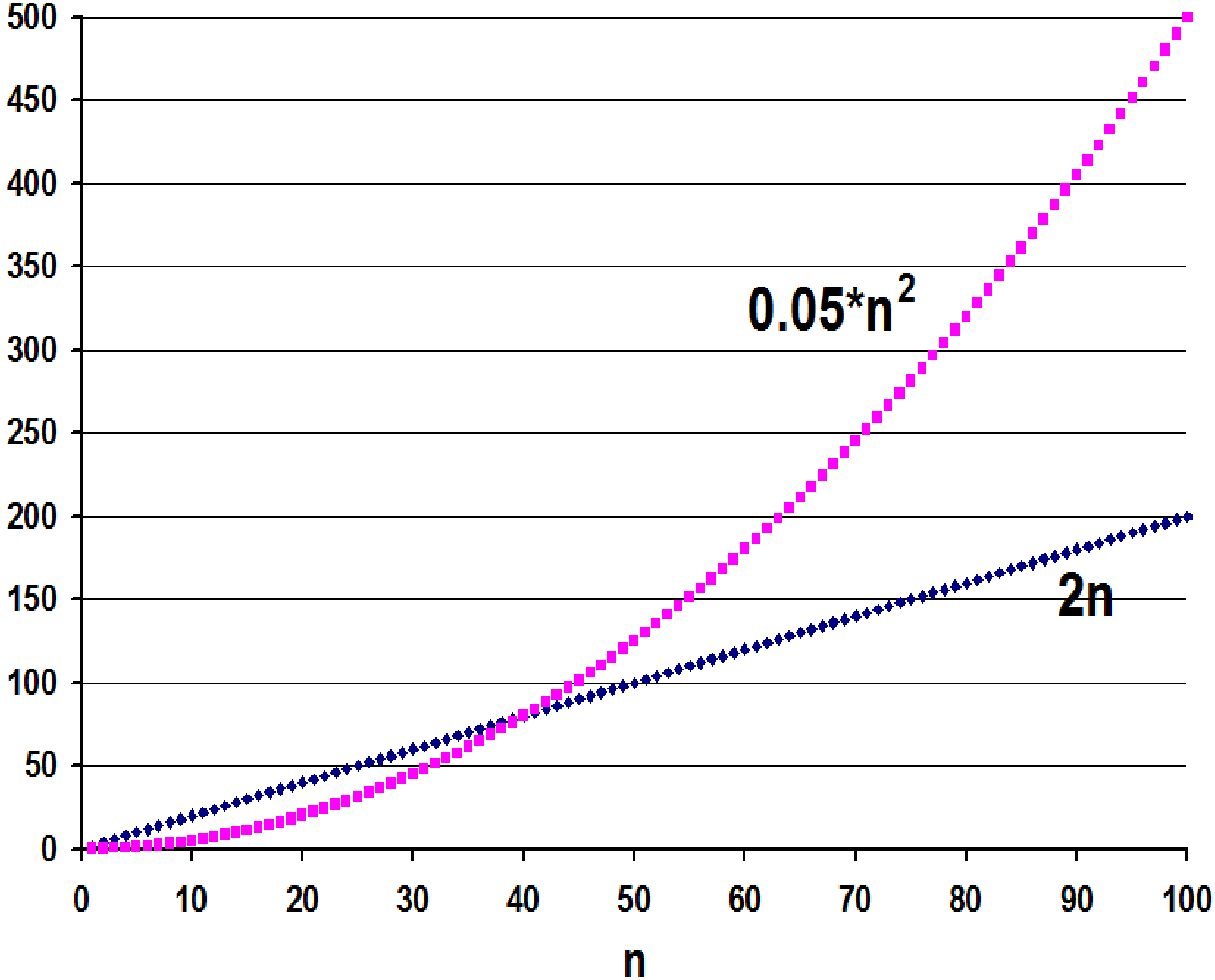
c1 and c2 are positive constants (> 0)
and n_0 is a non-negative integer

Thumb Rule for using Big-O and Big-Θ

- We say a function f(n) = O(g(n)) if the rate of growth of g(n) is either at the same rate or faster than that of f(n).
  - If the functions are polynomials, the rate of growth is decided by the degree of the polynomials.
  - Example: $2n^2 + 3n + 5 = O(n^2)$;
    $2n^2 + 3n + 5 = O(n^3)$;
  - note that, we can also come up with innumerable number of such functions for what goes inside the Big-O notation as long as the function inside the Big-O notation grows at the same rate or faster than that of the function on the left hand side.

- We say a function f(n) = Θ(g(n)) if both the functions f(n) and g(n) grow at the same rate.
  - Example: $2n^2 + 3n + 5 = \Theta(n^2)$ and not $\Theta(n^3)$;
  - For a given f(n), there can be only one function g(n) that goes inside the Θ-notation.

Asymptotic Notations: Example



$2n \le 0.05\ n^2$

for $n \ge 40$

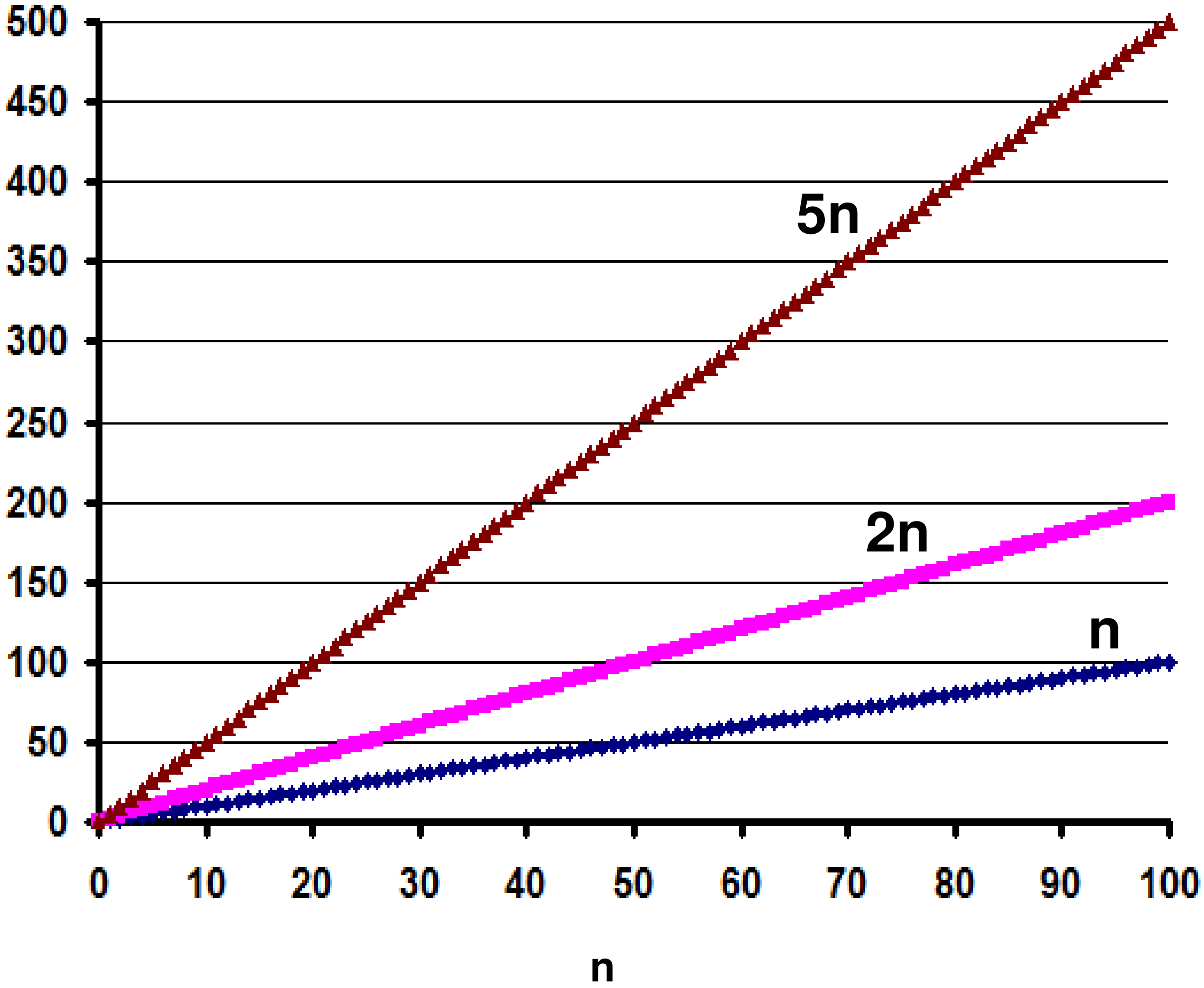
$c = 0.05$, $n_0 = 40$

$2n = O(n^2)$

More generally,

$n = O(n^2)$.

Asymptotic Notations: Example



for $n \geq 1$

$n \leq 2n \leq 5n$

$2n = \Theta(n)$

Relationship and Difference between Big-O and Big-Θ

- If $f(n) = \Theta(g(n))$, then $f(n) = O(g(n))$.
- If $f(n) = O(g(n))$, then $f(n)$ need not be $\Theta(g(n))$.
- Note: To come up with the Big-O/Θ term, we exclude the lower order terms of the expression for the time complexity and consider only the most dominating term. Even for the most dominating term, we omit any constant coefficient and only include the variable part inside the asymptotic notation.
- Big-Θ provides a tight bound (useful for precise analysis); whereas, Big-O provides an upper bound (useful for worst-case analysis).
- Examples:

  (1) $5n^2 + 7n + 2 = \Theta(n^2)$
  - Also, $5n^2 + 7n + 2 = O(n^2)$

  (2) $5n^2 + 7n + 2 = O(n^3)$,

  Also, $5n^2 + 7n + 2 = O(n^4)$, But, $5n^2 + 7n + 2 \neq \Theta(n^3)$ and $\neq \Theta(n^4)$
- The Big-O complexity of an algorithm can be technically more than one value, but the Big-Θ of an algorithm can be only one value and it provides a tight bound. For example, if an algorithm has a complexity of $O(n^3)$, its time complexity can technically be also considered as $O(n^4)$.

When to use Big-O and Big-Θ

- If the best-case and worst-case time complexity of an algorithm is guaranteed to be of a certain polynomial all the time, then we will use Big-Θ.
- If the time complexity of an algorithm could fluctuate from a best-case to worst-case of different rates, we will use Big-O notation as it is not possible to come up with a Big-Θ for such algorithms.

- **<u>Sequential key search</u>**
- **Inputs: Array A[0…n-1], Search Key K**
- **Begin**

```
        for (i = 0 to n-1) do
        if (A[i] == K) then
              return "Key K found at index i"
        end if
    end for
    return "Key K not found!!"
End
```

O(n) only and not Θ(n)

- **<u>Finding the Maximum Integer in an Array</u>**
- **Input: Array A[0…n-1]**
- **Begin**

```
      Max = A[0]
        for (i = 1 to n-1) do
        if (Max < A[i]) then
             Max = A[i]
        end if
    end for
    return Max
End
```

Θ(n) ➔It is also O(n)

Another Example to Decide whether Big-O or Big-Θ

Skeleton of a pseudo code

```
Input size: n
Begin Algorithm
If (certain condition) then
     for (i = 1 to n) do
       print a statement in unit time
   end for
else
     for (i = 1 to n) do
             for (j = 1 to n) do
                print a statement in unit time
             end for
     end for
End Algorithm
```

Best Case
The condition in the if block is true
-- Loop run 'n' times

Worst Case
The condition in the if block is false
-- Loop run 'n^2' times

Time Complexity: $O(n^2)$
It is not possible to come up with a Θ-based time complexity for this algorithm.

Data processed by an Algorithm

- The design and development as well as the time and storage complexities of an algorithm for a problem depend on how we store and process the data on which the algorithm is run.
- For example: if the words in a dictionary are not sorted, it would take a humongously long time to come up with an algorithm to search for a word in the dictionary.
- Sometimes, the data need not be linear (like a dictionary) and need to be hierarchical (like a road map or file system).
- Layman example
  - Abstract view of a car (any user should expect these features for any car): Should be able to start the car, turn steering, press brake to stop and press gas to accelerate, change gear, etc.
  - Implementation (responsibility of the manufacturer and not the user): How each of the above is implemented? Varies with the targeted gas efficiency, usage purpose, etc.

Abstract Data Type (ADT) vs. Data Structures

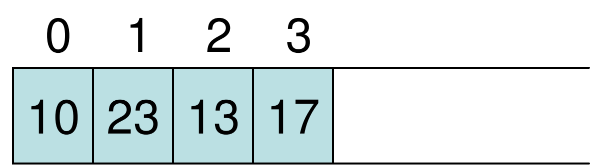
- Data processed by an algorithm could be represented at two levels:
  - Abstract level (also called logical or user level): merely state the possible values for the data and what operations/functions the algorithm will call to store and access the data
  - Implementation level (also called system level): deals with how the implementation should be done to perform the functions defined for the data at the abstract level.
- The abstract (logical) representation of data is commonly referred to as Abstract Data Type (ADT)
- The term "data structure" is considered to represent the implementation model of an ADT.

Common ADTs and the Data Structures for their Implementation

- List, Stack, Queue
  - Arrays, Linked List
- Priority Queue
  - Heap
- Dictionary
  - Hash Table, Binary Search Tree
- Graph
  - Adjacency List, Adjacency Matrix

List ADT

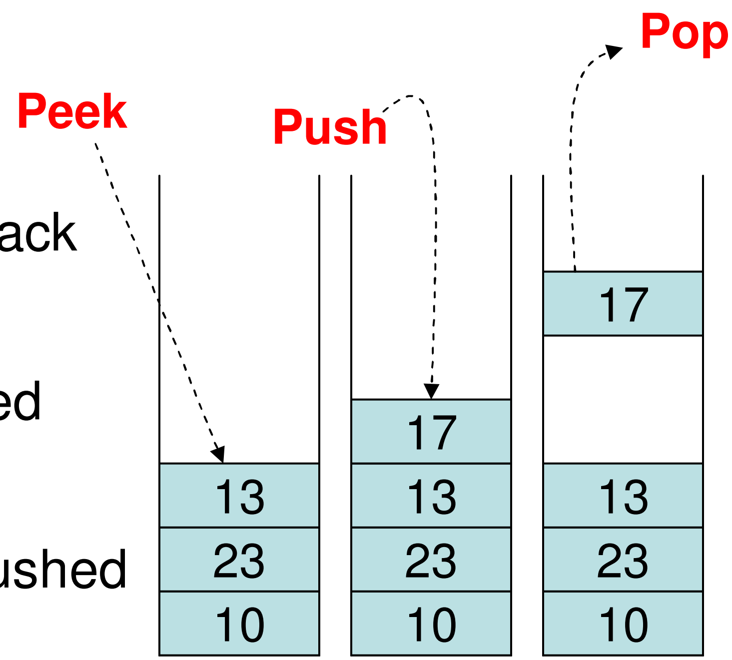
- Data type
  - Store a given number of elements of any data type



- Functions/Operations
  - Create an initial empty list
  - Test whether or not a list is empty
  - Read element based on its position in the list.
  - Insert, delete or modify an entity at a specific position in the list

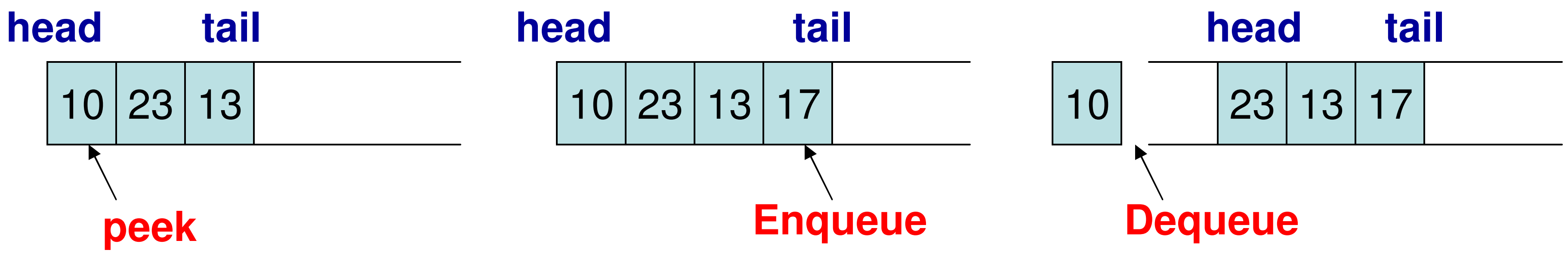
Stack ADT

- Data type
  - Store a given number of elements of any data type
- Unique characteristic: Last In First Out (LIFO)
- Functions/Operations
  - Insert
    - Push an element to the top of the stack
  - Delete
    - Pop the last element that was pushed
  - Read
    - Peek at the last element that was pushed
  - Check if empty



Queue ADT

- Data type
  - Store a given number of elements of any data type
- Unique characteristic: First In First Out (FIFO)
- Functions/Operations
  - Insert
    - Enqueue: Append an element to the end of queue
  - Delete
    - Dequeue: Remove the element at the head of the queue
  - Read
    - Peek: Look at the element at the head of the queue
  - Check if empty



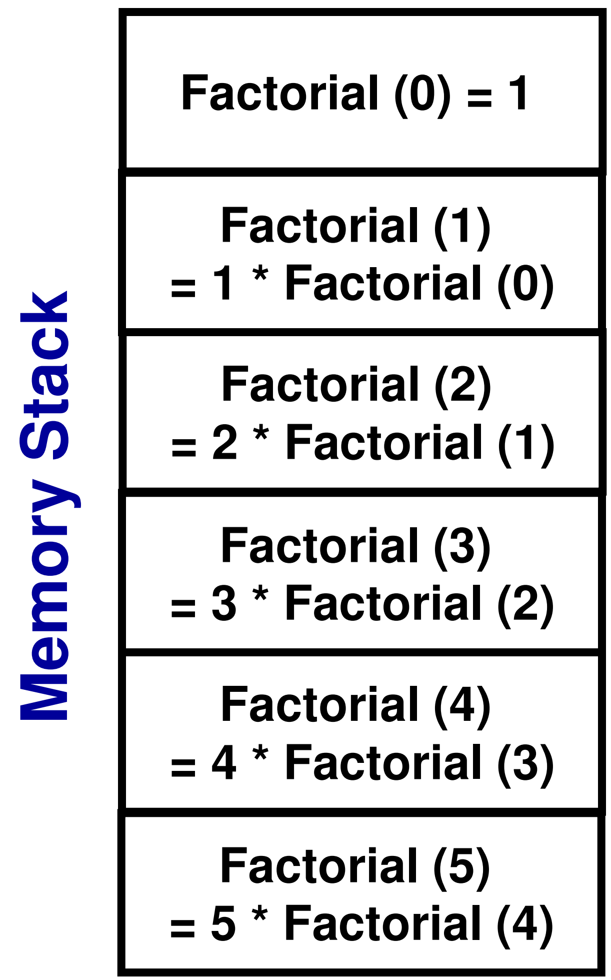
Recursion

- Recursion: A function calling itself.
- Recursions are represented using a recurrence relation (incl. a base or terminating condition)

- Example 1
- Factorial (n) = n * Factorial(n-1) for n > 0
- Factorial (n) = 1 for n = 0

```
Factorial(n)
  if (n == 0)
      return 1;
  else
      return n * Factorial(n-1)
```

Factorial(0) = 1
Factorial(1) = 1 * Factorial(0)
Factorial(2) = 2 * Factorial(1)
Factorial(3) = 3 * Factorial(2)
Factorial(4) = 4 * Factorial(3)
Factorial(5) = 5 * Factorial(4)



Recursion

- Example 2: Fibonacci Series
- F(n) = F(n-1) + F(n-2) for n > 1
- F(0) = 0 and F(1) = 1

```
F(n)
   if (n == 0)
       return 0;
   else if (n == 1)
       return 1;
   else
       return F(n-1) + F(n-2)
```