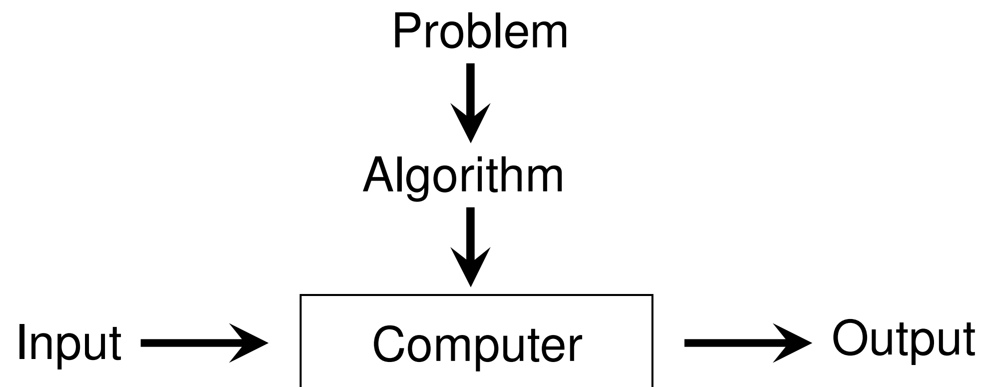
Module 1: Asymptotic Time Complexity C++ Review

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

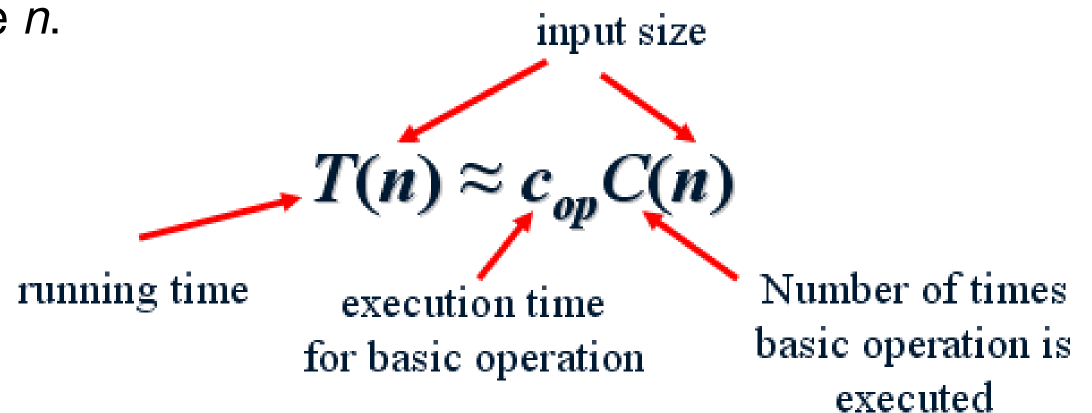
The Analysis Framework

- **Time efficiency (time complexity):** indicates how fast an algorithm runs
  - The time complexity of an algorithm is typically represented as a function of the input size
    - E.g., sorting an array of ‘n’ integers, traversing a graph of ‘V’ vertices and ‘E’ edges
  - If the input is just one element, the time complexity is represented as function of the number of bits needed to represent the input.
    - E.g., log(n) to determine whether an integer ‘n’ is prime or not.

- **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 whose space complexity does not increase with input size (i.e., requires the same additional space irrespective of input size) are said to be ***in-place***.

Units for Measuring Running Time

- The theoretical 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
```

0	1	2	3	4	5	6	7
5	7	3	1	8	10	2	9

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.
- <u>Algorithm A (naïve)</u>

```
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

<u>Algorithm B (optimal)</u>

```
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: $\sqrt{n} - 2 + 1$
= $\sqrt{n} - 1$ divisions
For larger n: $\sqrt{n}$

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

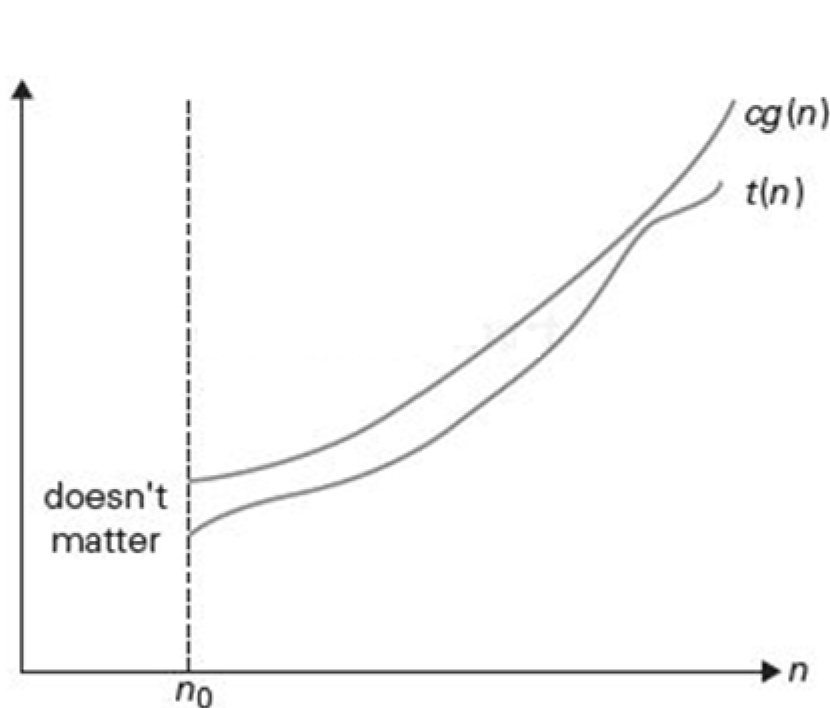
- We are more interested in the order of growth on the number of times the basic operation is executed on the input size of an algorithm.
- We focus on the asymptotic order of growth: i.e., what happens when the input size (n) grows larger.
- For example, if the number of basic operations of two algorithms 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.

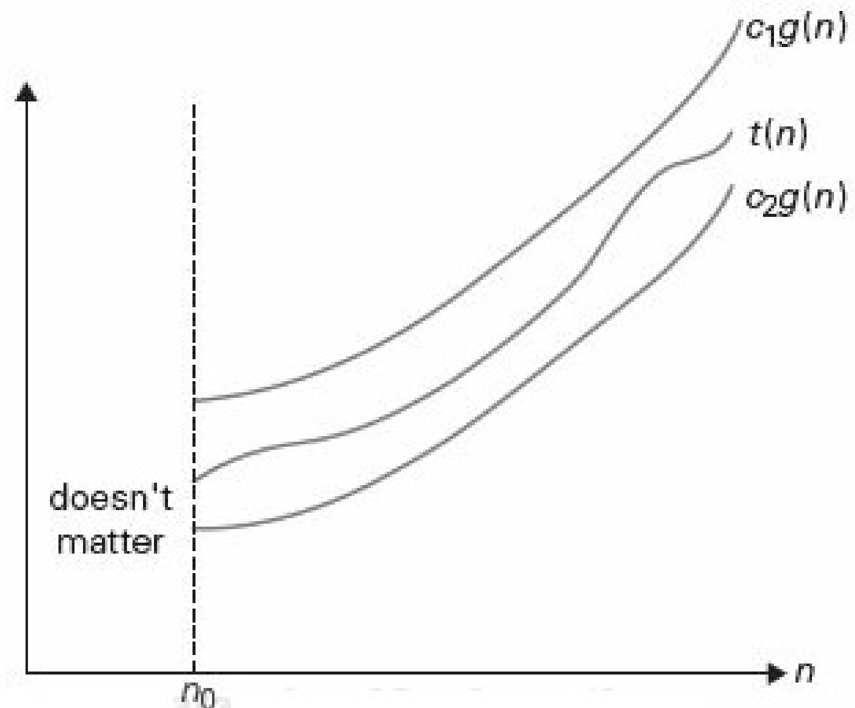
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$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





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

$t(n) \leq c*g(n)$ for all $n \geq n_0$

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

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

$c2*g(n) \leq t(n) \leq c1*g(n)$ for all $n \geq n_0$

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

Thumb Rule for using Big-O and Big-Θ

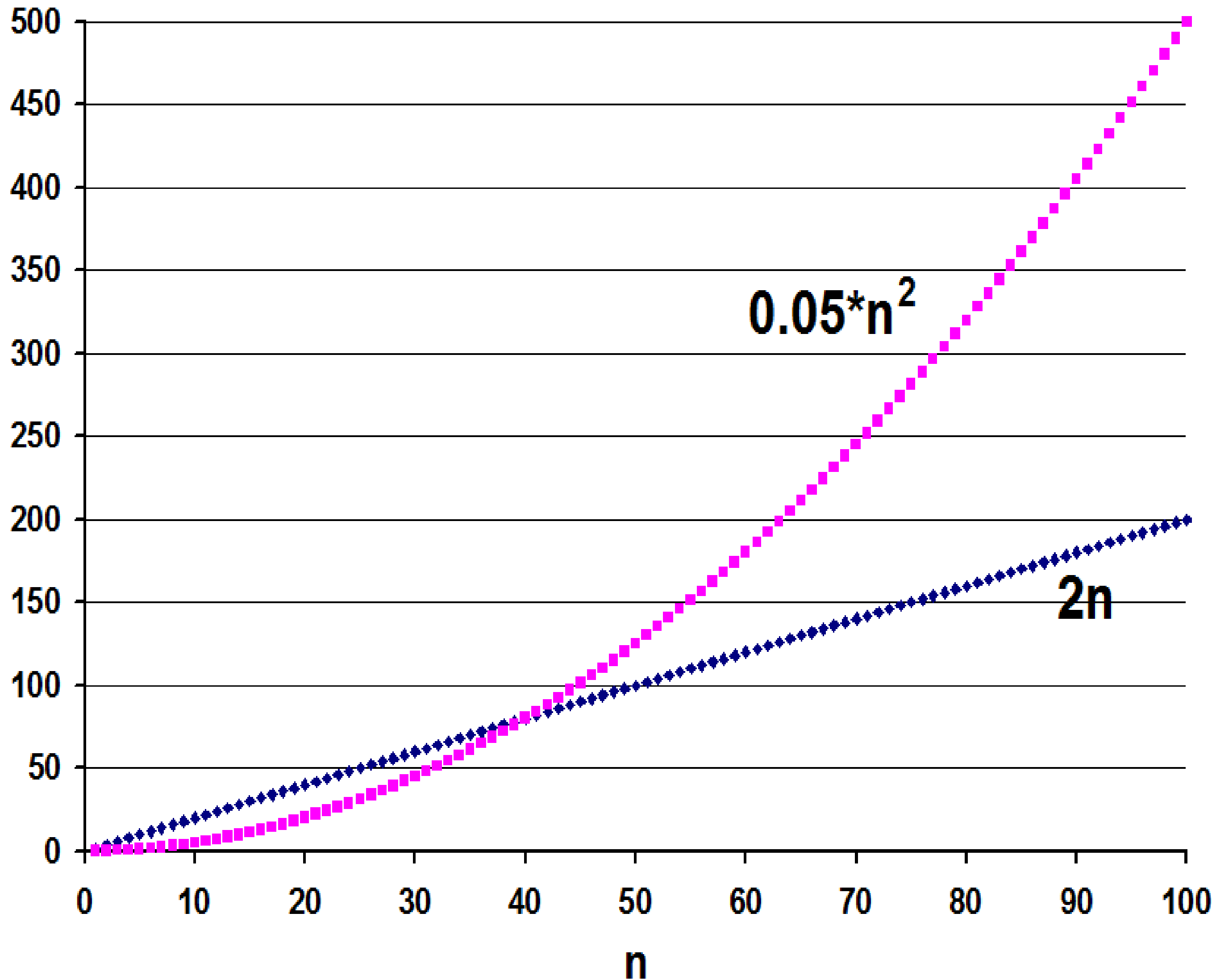
- We say a function $f(n) = O(g(n))$ if the rate of growth of $g(n)$ is either at the same rate or faster than that of $f(n)$.
  - If the functions are polynomials, the rate of growth is decided by the degree of the polynomials.
  - Example: $2n^2 + 3n + 5 = O(n^2)$;
    $2n^2 + 3n + 5 = O(n^3)$;
  - 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.

'Faster' means value of the function quickly increases with increase in 'n'

- We say a function $f(n) = \Theta(g(n))$ 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.

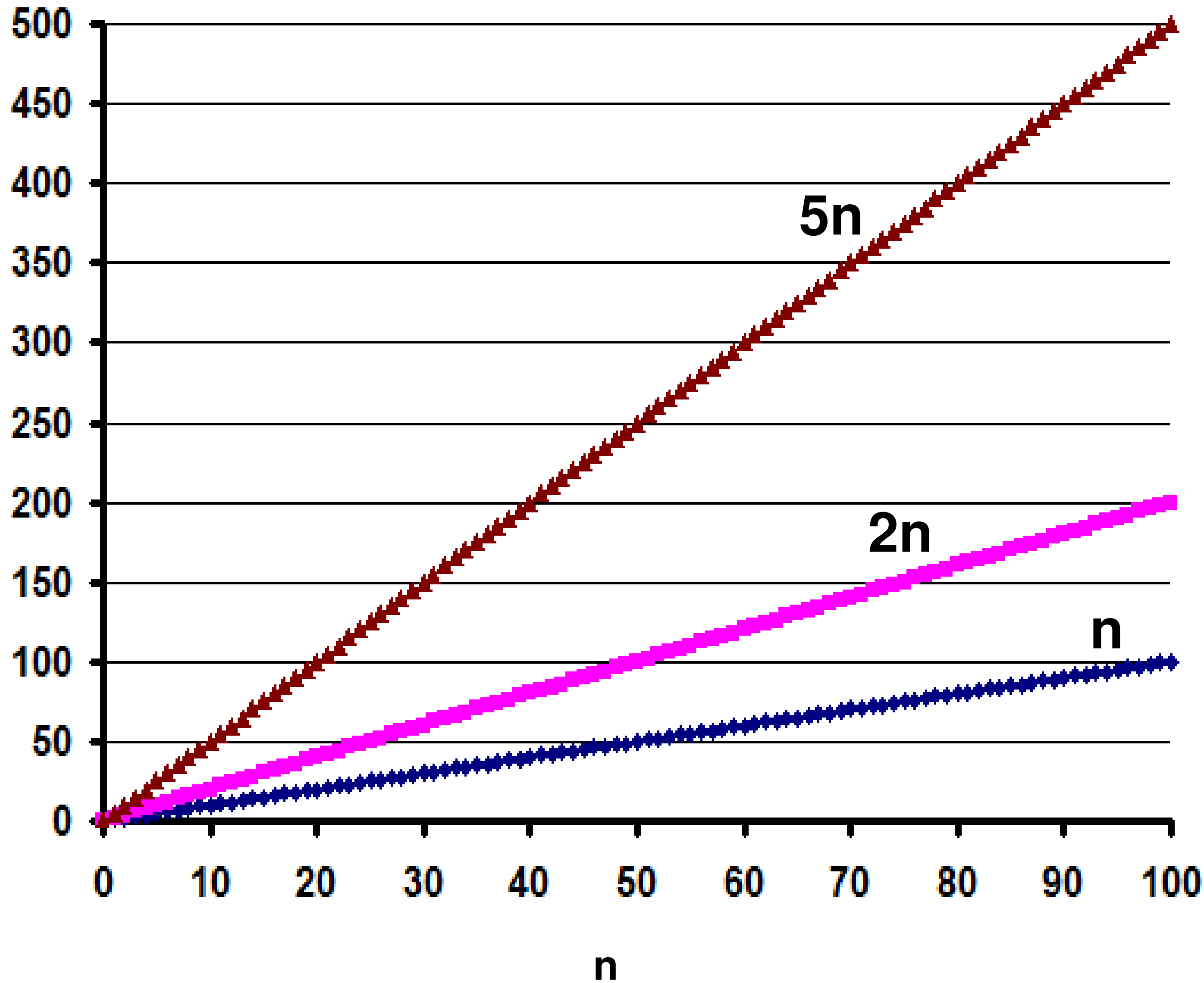
O – Loose Bound **Θ – Tight Bound**

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
```

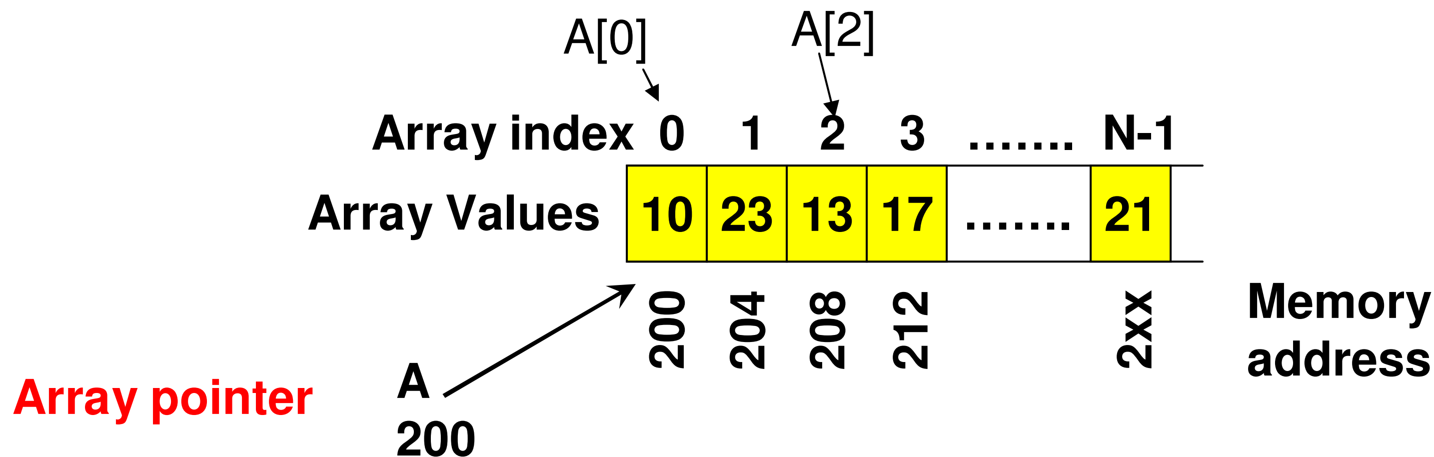
Best Case
The condition in the if block is true
-- Loop run 'n' times

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.

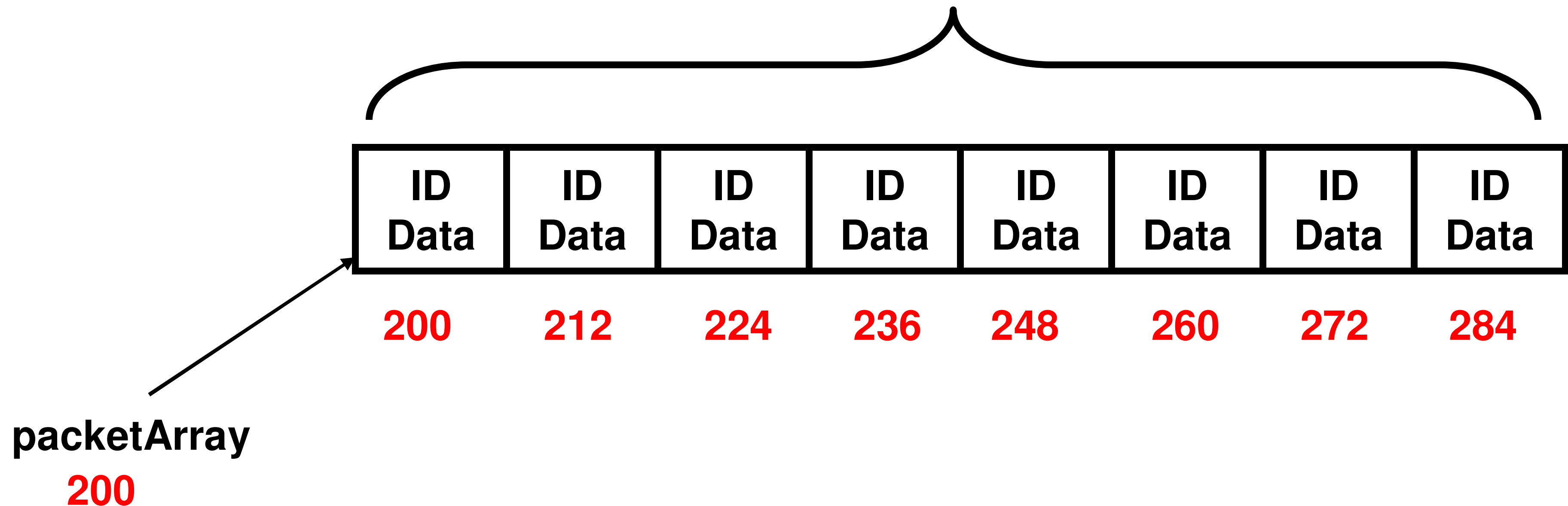
Arrays

- When we create an array (say, of integers) of size N, the operating system will allocate memory space that can be used to store the 'N' integers in consecutive blocks of memory.
- The starting address of the allocated memory block will be stored in a pointer to the array.
- Any index in an array can be accessed in constant time (i.e., Θ(1) time). For example, to access A[i], all we have to do is take the base address of the array pointer and to it the product of the array index *i* and the # bytes for storing the data type in the array.
- In the case of integers (example below), the address of the element at A[2] is 200 + 2*4 = 208. We thus do not need sequential access.
- To access A[9], we just need to do 200 + 9*4 = 236 and access the element there.



Arrays of Objects

```
Packet* packetArray = new Packet[numPackets];
        for (int id = 0; id < numPackets; id++){
                packetArray[id].setID(id);
                packetArray[id].setData(id*10);
        }
```



C++ Code Review

Example 1: Arrays, Pointers and Random Number Generator

```
#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;
```

```
int main(){

    int arraySize;
    cout << "Enter the array size: ";
    cin >> arraySize;

    int maxValue;
    cout << "Enter the maximum value for an element: ";
    cin >> maxValue;

    int *array = new int[arraySize];

    srand(time(NULL));

    for (int i = 0; i < arraySize; i++){

        array[i] = 1 + rand() % maxValue;

    }
```

```cpp
    cout << "Sum of all elements is " << addALL(array, arraySize) << endl;

    int *negArray = TurnNegative(array, arraySize);

    cout << "Negative Values of the Elements " << endl;

    for (int i = 0; i < arraySize; i++)
        cout << negArray[i] << " ";

    cout << endl;

    delete[] array;
    delete[] negArray;


return 0;

}
```

```cpp
int addALL(int* Array, int ArraySize){

    int sum = 0;

    for (int i = 0; i < ArraySize; i++)
        sum += Array[i];

    return sum;

}
```

```
int* TurnNegative(int* Array, int ArraySize){

    int* negativeArray = new int[ArraySize];

    for (int i = 0; i < ArraySize; i++)
        negativeArray[i] = -1*Array[i];

    return negativeArray;

}
```

C++ Code Review

Example 2: Class, Pointers with Objects