# Efficiency of Algorithms

ESC108A Elements of Computer Science and Engineering
B. Tech. 2017


**Course Leaders:**

**Roopa G.**

**Ami Rai E.**

**Chaitra S.**

# Objectives

- At the end of this lecture, student will be able to
  - Explain the terms ‘efficiency’ and ‘complexity’
  - Calculate efficiency of an algorithm
  - Express the complexity of algorithms in asymptotic notation
  - Classify algorithms based on their complexity

# Contents

- Efficiency and Complexity
- Time Complexity of an algorithm
- Space Complexity of an algorithm
- Measuring complexity of a sequential algorithm
- Measuring complexity of an algorithm with branching
- Measuring complexity of an algorithm with loops

# Computer Engineering

## Develop Good Quality Systems

Build ***Stable*** Software and Hardware

Build ***Efficient*** Software and Hardware

# Which is Better Algorithm?

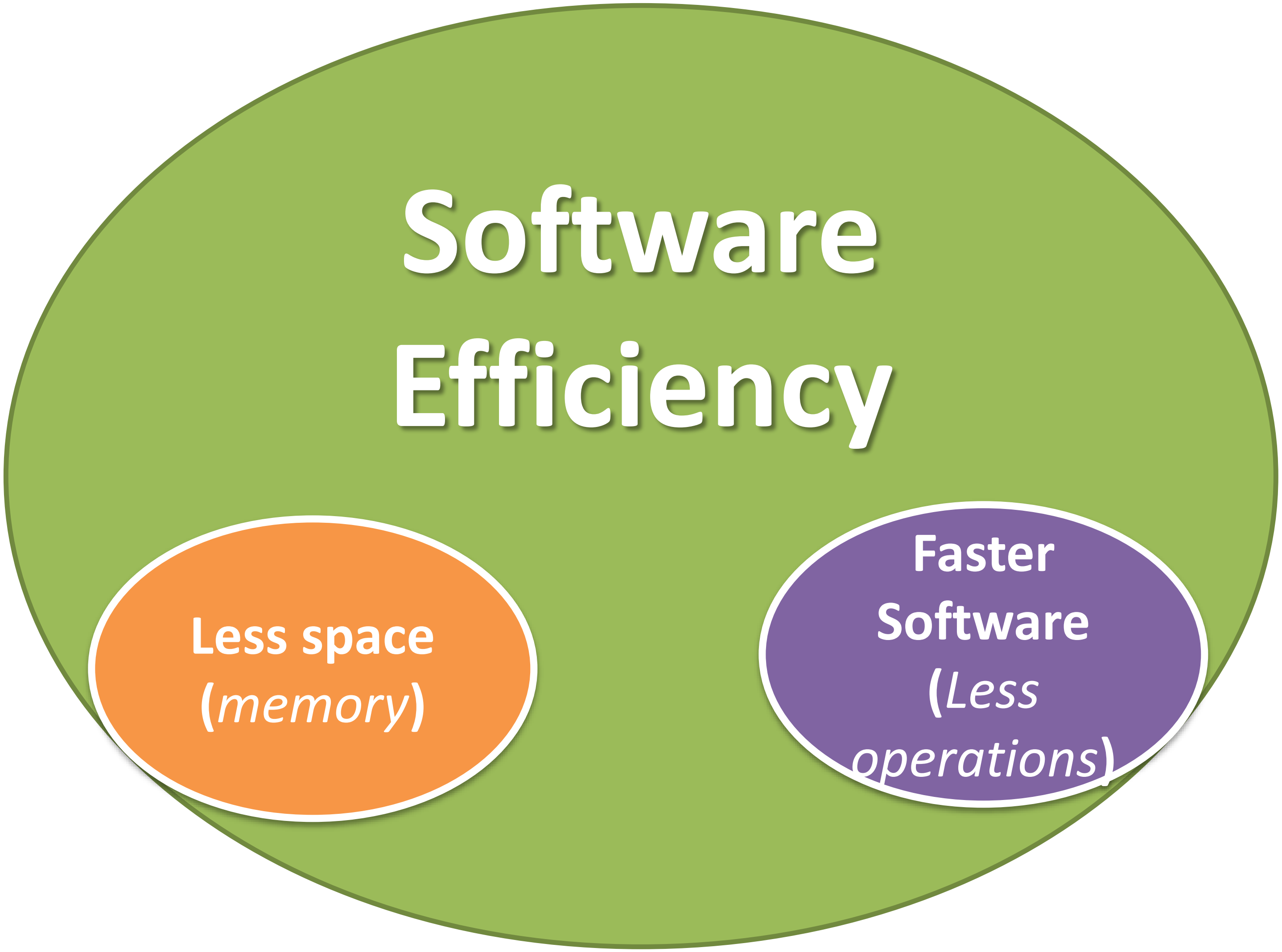
```
Algorithm multiply1 (var m, n: Integer): Integer
var index, temp: Integer;
begin
        if (n=0) then
        begin
                temp := 0;
        end
        else
        begin
            temp := m;
            for index := 2 to n do
            begin
                temp := temp + m;
            end
        end
end
```

```
Algorithm multiply2 (var m, n : Integer): Integer
var temp: Integer;
begin
        temp := m * n;
end
```

Why

- Less space
- Less time
- More stable

# Software Efficiency



# Complexity of an Algorithm

- Time Complexity
  - Time is a factor in measuring the efficiency of computer program
  - A measure of time taken for an algorithm to execute
    - Number of cycles
- Space Complexity
  - Space is also a factor in measuring the efficiency of computer program
  - A measure of space taken for executing an algorithm
    - Number of words

# Complexity of a Sequential Algorithm

- An Example: Swap 2 numbers

```
Algorithm swap
var a,b,temp : Integer;{Space complexity: 3 words}
begin
    temp := a; {Time complexity: 1 cycle}
    a := b; {Time complexity: 1 cycle}
    b := temp; {Time complexity: 1 cycle}
end
```

Total Space complexity: 3 ***words***

Total Time complexity: ***3 cycles***

# Complexity of an Algorithm with Branching

```
Algorithm isEven (var a:Integer) { Space complexity: 1 word}
var ret: boolean; { Space complexity: 1 word}
begin
    {assert a > 0} {Time complexity 1 cycle}
    if ( (a mod 2) = 0) then {Time complexity 1 cycle}
    begin
            ret := true; {Time complexity 1 cycle}
    end
    else
    begin
            ret := false;
            {Time complexity 1 cycle}
    end
end
```

Total Space complexity: 2 ***words***

Total Time complexity:

2 cycles + {1 cycle or 1 cycle}

= 3 ***cycles***

# Complexity of an Algorithm with Looping

```
Algorithm factorial (var n:Integer) { Space complexity: 1 word}
var ret, iLoop: Integer;  { Space complexity: 2 words}
begin
        {assert n > = 0}  {Time complexity 1 cycle}
        ret := 1;  {Time complexity 1 cycle}
        for iLoop in 2 to n do  {Time complexity n cycles+ 1 cycle}
        begin
                ret := ret * iLoop;  {Time complexity 2 cycle}
        end
end
```

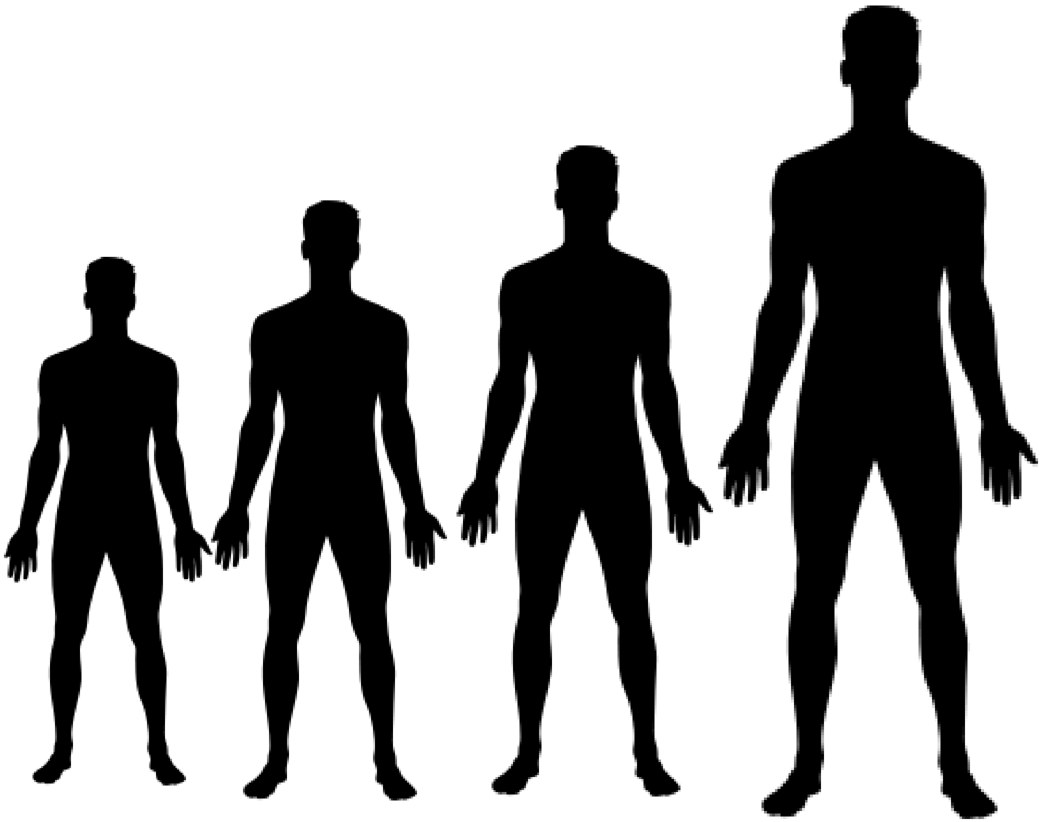
Total Space complexity: 3 ***words***

Total Time complexity:

3 cycles + n times 2 cycles

= 2n+3 ***cycles***

# Growth Rate



# Growth Rate

**Algorithm** *sumN(array:nIntegerElements, n:Integer)*:Integer;

**var** s,i:**Integer**; {The partial sum}

**begin**

    s := 0; {1 cycle}

    for i in 0 **to** n **do** {n times + 1 cycle}

    **begin**

        s := s + array[i]; {2 cycle}

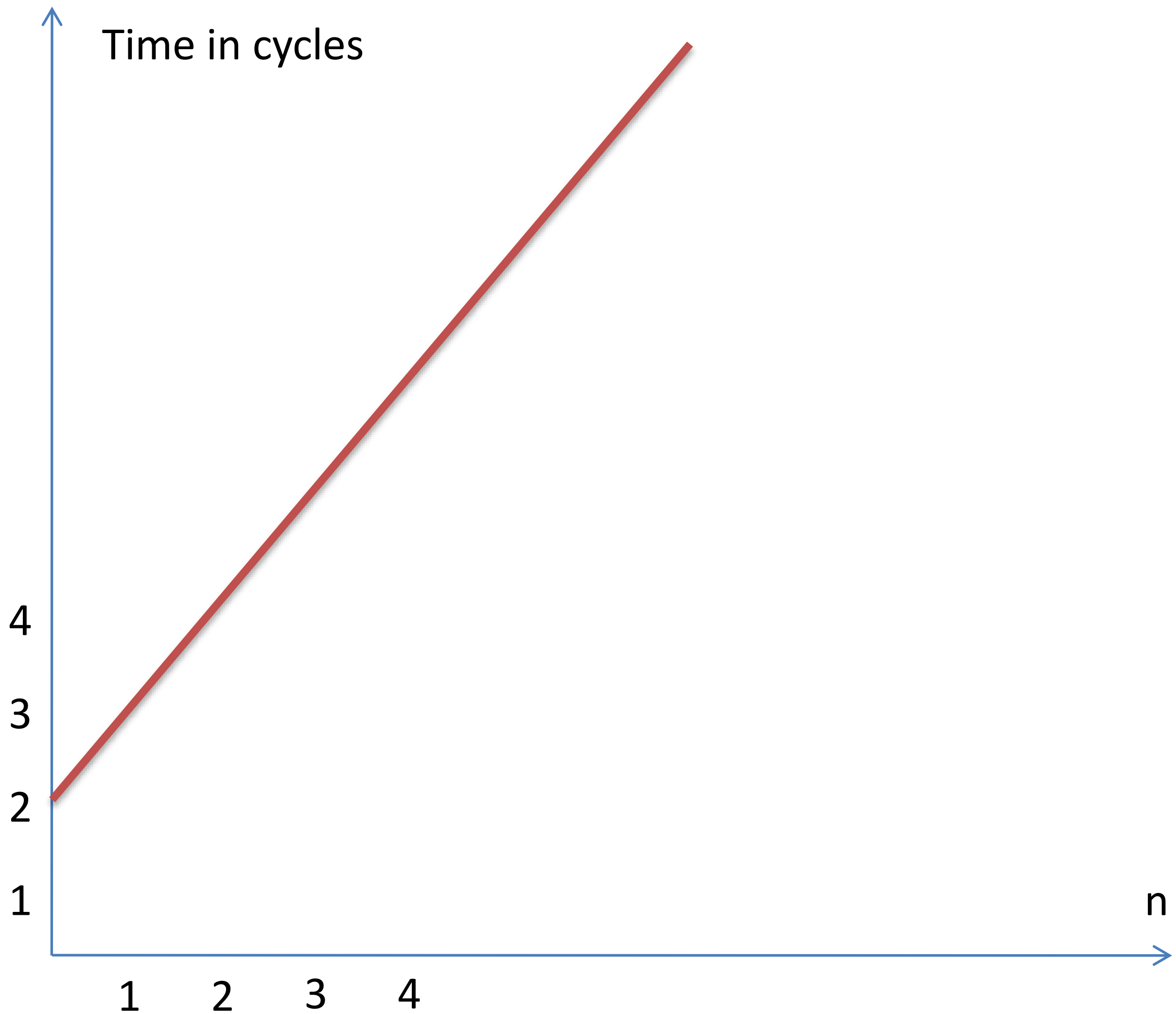
    **end**

**end**

**Total time = 1+n*2+1 = 2n+2 cycles**

How much time will this algorithm take when the value of n is increased from 0 towards infinity in steps of 1?

# Growth Rate



# Examples

- An Example: Swap 2 numbers

**Algorithm** *swap* (**var** a,b : **Integer**) { Space complexity: 2 words}

**var** temp : **Integer**;{Space complexity: 1 word}

**begin**

temp := a; {Time complexity: 1 cycle}

a := b; {Time complexity: 1 cycle}

b := temp; {Time complexity: 1 cycle}

**end**

***Type of time complexity:*** Constant (does not change with different inputs)

***Type of space complexity:*** Constant

# Examples

```
Algorithm isEven (var a:Integer) { Space complexity: 1 word}
var ret: boolean; { Space complexity: 1 word}
begin
    {assert a > 0} {Time complexity 1 cycle}
    if ( (a mod 2) = 0) then {Time complexity 1 cycle}
    begin
            ret := true; {Time complexity 1 cycle}
    end
    else
    begin
            ret := false; {Time complexity 1 cycle}
    end
end
```

***Type of time complexity:*** Constant

***Type of space complexity:*** Constant

# Examples

**Algorithm** *factorial* (**var** n:**Integer**) { Space complexity: 1 word}

**var** ret, iLoop: **Integer;** { Space complexity: 2 words}

**begin**

    **{assert** n > = 0**}** {Time complexity 1 cycle}

    ret := 1; {Time complexity 1 cycle}

    **for** iLoop **in** 2 to n **do** {Time complexity n cycles}

    **begin**

        ret := ret * iLoop; {Time complexity 1 cycle}

    **end**

**end**

***Type of time complexity:***

Linear

***Type of space complexity:***

Constant

# Asymptotic Analysis

- Asymptotic analysis of algorithms
  - Describe behavior of algorithms with bounds
  - A way to group algorithms having similar performance behavior
- Big-Oh (O) notation is the most popular as it provides an upper bound to the behavior of an algorithm
  - O(f(x)) tells that the complexity of the algorithm is always limited with f(x)+c1 as upper bound, where c1 is an arbitrary constant
- For example:
  - 2 cycles and 3 cycles all belong to constant time complexity - O(1)
  - n cycles, n+2 cycles, 2n+3 cycles all belong to linear time complexity – O(n)

# Summary

- Efficiency is the process of achieving maximum productivity with minimum wasted effort or expense
- For algorithms, efficiency is measured in terms of space (memory used) and time (number of operations) complexity
- Space and time complexity are estimated by expressing as growth rate functions
- The upper bound of worst case growth rate is generally used to categorise algorithms and the notation used is Big-Oh notation (O)
- If the steps are in sequence, the time complexity is the sum of the steps
- If there is a branch based on a condition, the time complexity is 1+worst case branch complexity

# Summary contd.

- If there is looping for 'n' times on a block, the complexity of the block is multiplied by n times and 1 extra cycle is added for the last check
- Always consider the maximum complexity of the function

# References

Toida, S. (2013) Summary of Big-Oh, *Growth Functions, available at http://www.cs.odu.edu/~toida/nerzic/content/function/growth.html (accessed 22 July 2014).*

# Further Reading

Horowitz, E. And Sahni, S. (1983). Chapter 1: Introduction, *Fundamentals of Data Structures*, Pitman.