AUBURN

UNIVERSITY

Growth of Functions

Overview

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

Objectives

- Learn and understand how to characterize the efficiency of algorithms.
- Learn, understand, and use asymptotic notation.
- Learn, understand, and use growth of functions to characterize efficiency (time and space complexity).

Requirements

- Review material covered in *CPSC 3243 Discrete Structures* or equivalent
- Read
  - Introduction of Chapter 3,
  - Section 3.1 (Θ notation, O notation, and Ω notation)
  - Section 3.2 (Floors and ceilings, modular arithmetic, polynomials, exponentials, logarithms, factorials) Not covered here

How to Characterize Running Time/ Time complexity?

- What are the key parameters of the running time/time complexity?
- How to characterize the running time/time complexity f(n) based on the intrinsic "behavior" of an algorithm?
  - **answer**: asymptotic growth of f(n)

Key Parameters of the Running Time of An Algorithm

- Consider a program expressed in some high-level language (e.g., pseudocode):

```
max(A)
    m = A[1]
    i = 2;
    while (i ≤ A.length)
        if (m < A[i])
            m = A[i]
        i++
    return m
```

- The running time (or time complexity) will depend on:
  1. the number of (pseudocode) instructions expressing the algorithm → Algorithm itself
  2. the number of **machine (CPU)** instructions per high-level instruction → Compiler/Instruction set
  3. the number of clock cycles per machine instruction → CPU
  4. the duration of a clock cycle → Clock frequency, CPU
  5. and…... the input size
- How are these parameters determined?

How to INTRINSICALLY Characterize the Running Time?

- **Objective**: characterize the running time/complexity time based **only** on:
  - the algorithm itself
  - the input size ***n***, specifically when n is *large* (tending to infinity)
- In other words, we want to characterize the time complexity ***independently*** of the compiler, the instruction set, the CPU, or the clock.
- If the function *f(n)* represents the time (or space) complexity of an algorithm, we will be interested in the **asymptotic** growth of f(n), rather than specific values of f(n) for values of n. We are interested in f(n) with **very large value**s of n (input size).
- Let us review an example we already covered. But, this time we will justify the notion of "*growing as* n^2".

Growth of Functions

- Consider these four functions:

$$f_1(n) = 10\ln(n)$$

$$f_2(n) = 5.n^2$$

$$f_3(n) = n^5$$

$$f_4(n) = 2^n$$

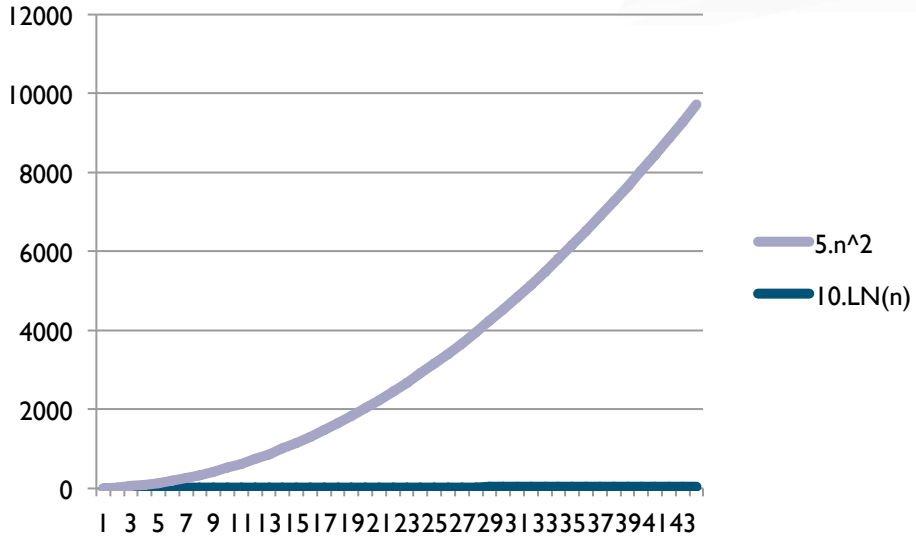
- We are interested in the behavior of these functions for **very large** values on n.

- Which function grows faster and higher?

$f_1(n) = 10\ln(n)$ versus $f_2(n) = 5.n^2$

Observe

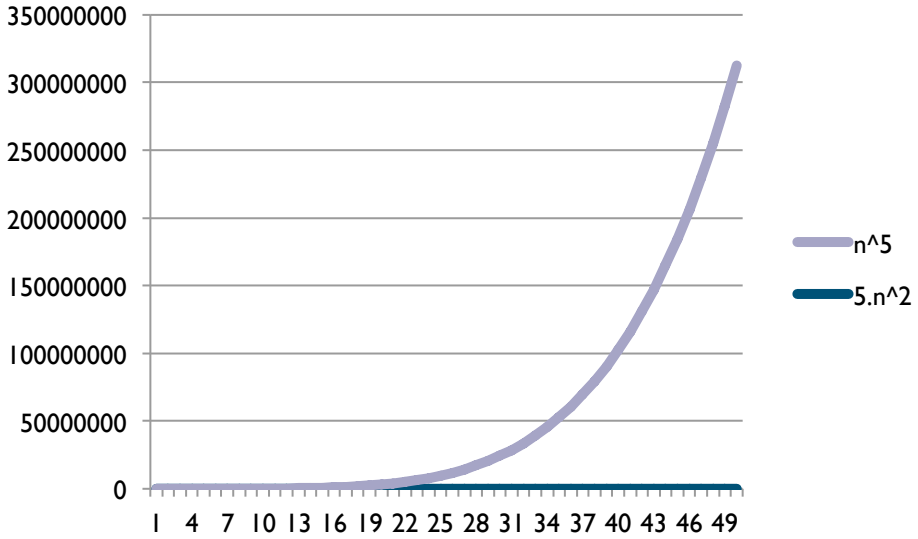
1. the range: highest *y*-value is about 10,000
2. how f_2 dwarfs f_1 starting at n = 15…



$f_2(n) = 5.n^2$ versus $f_3(n) = n^5$

Observe

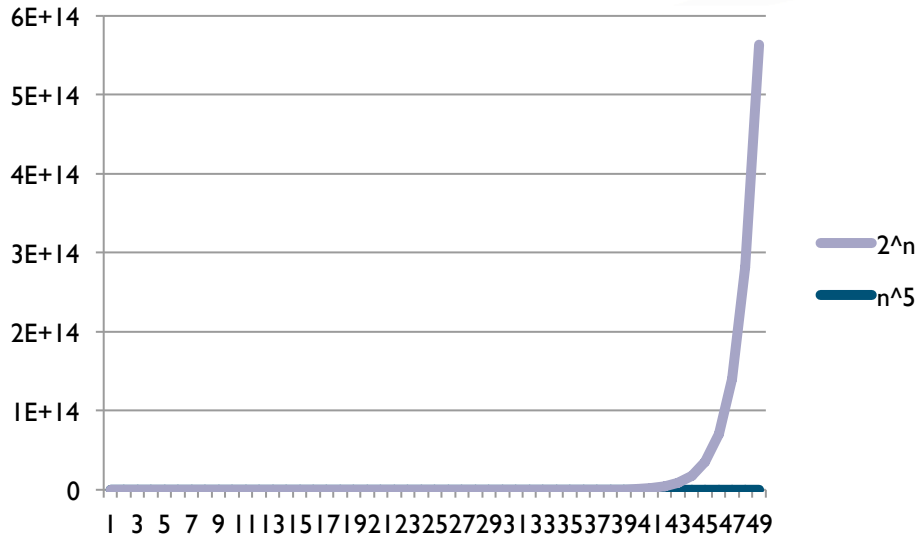
1. the range: highest *y*-value is about 350,000,000
2. how f_3 dwarfs f_2 starting at n = 35…



$f_3(n) = n^5$ versus $f_4(n) = 2^n$

Observe

1. the range: highest *y*-value is about 5.5 E^{+14}
2. how f_4 dwarfs f_3 starting at n = 47…



Conclusions About Functions Growth

- f_1, f_2, f_3, and f_4 are very different "animals" in terms of growth
  - f_1 : log function (ln(n))
  - f_2 and f_3: n^c polynomial functions
  - f_4 : $2^n = e^{n\ln(2)}$ exponential functions
- When evaluating time or space complexity, it will be sufficient just to determine which "animal": log, polynomial, or exponential.
- If two functions are *similar* "animals", then we can look at the *details* (coefficients):
  - $f_2=5n^2$ and $f_3=n^5$ are similar "animals": both are polynomials. Then we look at the powers 2 and 5 to compare them.

How to Characterize Running Time/ Time complexity?

- Review the time complexity of the naïve sorting algorithm
- How to characterize the running time/time complexity f(n) based on the intrinsic "behavior" of an algorithm?
  - **Answer**: the asymptotic behavior of f(n)

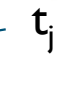
Time Complexity of "*Sorting*" (1/3)

```
Sort-Array(A)
        for j = 1 to (A.length - 1)
                for i = (j + 1) to A.length
                        if (A[i] < A[j])
                                // swap A[i] and A[j]
                                buffer = A[j]
                                A[j] = A[i]
                                A[i] = buffer
```

- First, what is the **size of the input**?
  - **n**: length of the array, i.e, number of items in the array A (input is the array A)
- Which "action/operation"?
  - Action/operation: addition (j index update), swap, or comparison. Comparison seems more appropriate (comparison of index for the number of loop and comparision to determine the minimum)
- Let us count the number of comparisons and express them as a function of n.

Time Complexity of "*Sorting*" (2/3)

```
Sort-Array(A)
        for j = 1 to (A.length - 1)
            for i = (j + 1) to A.length
                if (A[i] < A[j])
                        // swap A[i] and A[j]
                        buffer = A[j]
                        A[j] = A[i]
                        A[i] = buffer
```



- Let us count the number f(n) of **comparisons** and express them as a function of n. This number f(n) is equal to the sum of:
  1. The outer loop (for j =1 ..) performs **n** comparisons (recall than A.length = n)
  2. The inner loop (for I = (j+1) ..) performs t_j comparisons. The number t_j depends on the value j:

     $j = 1, t_j = t_1 = n$

     $j = 2, t_j = t_2 = n - 1$

     $j = 3, t_j = t_3 = n - 2$

     ……

     $j = n-1, t_j = t_{n-1} = n - (n-1-1) = 2$
  3. The body of the inner loop performs s_j comparisons (A[i] < A[j]).

     $j = 1, s_j = s_1 = n - 1$

     $j = 2, s_j = s_2 = n - 2$

     $j = 3, s_j = s_3 = n - 3$

     ……

     $j = n-1, s_j = s_{n-1} = n - (n-1) = 1$

Time Complexity of "*Sorting*" (2/3)

```
Sort-Array(A)
    for j = 1 to (A.length - 1)
        for i = (j + 1) to A.length
            if (A[i] < A[j])
                // swap A[i] and A[j]
                buffer = A[j]
                A[j] = A[i]
                A[i] = buffer
```

- Let us count the number f(n) of comparisons and express them as a function of n. This number f(n) is equal to the sum of:
  1. The outer loop (for j =1 ..) performs **n** comparisons (recall than A.length = n)
  2. The inner loop (for I = (j+1) ..) performs t_j comparisons. The number t_j depends on the value j:

     j = 1, $t_j = t_1 = n$
     j = 2, $t_j = t_2 = n - 1$
     j = 3, $t_j = t_3 = n - 2$
     ……
     j = n-1, $t_j = t_{n-1} = n - (n-1-1) = 2$

     $$\sum_{j=1}^{n-1} t_j$$
  3. The body of the inner loop performs s_j comparisons (A[i] < A[j]).

     j = 1, $s_j = s_1 = n - 1$
     j = 2, $s_j = s_2 = n - 2$
     j = 3, $s_j = s_3 = n - 3$
     ……
     j = n-1, $s_j = s_{n-1} = n - (n-1) = 1$

Time Complexity of "*Sorting*" (2/3)

```
Sort-Array(A)
    for j = 1 to (A.length - 1)
        for i = (j + 1) to A.length
            if (A[i] < A[j])
                // swap A[i] and A[j]
                buffer = A[j]
                A[j] = A[i]
                A[i] = buffer
```

t_j

s_j

- Let us count the number f(n) of comparisons and express them as a function of n. This number f(n) is equal to the sum of:
  1. The outer loop (for j =1 ..) performs **n** comparisons (recall than A.length = n)
  2. The inner loop (for I = (j+1) ..) performs t_j comparisons. The number t_j depends on the value j:

     j = 1, $t_j = t_1 = n$

     j = 2, $t_j = t_2 = n - 1$

     j = 3, $t_j = t_3 = n - 2$

     ……

     j = n-1, $t_j = t_{n-1} = n - (n-1-1) = 2$

     $$\sum_{j=1}^{n-1} t_j = \sum_{k=2}^{n} k$$

  3. The body of the inner loop performs s_j comparisons (A[i] < A[j]).

     j = 1, $s_j = s_1 = n - 1$

     j = 2, $s_j = s_2 = n - 2$

     j = 3, $s_j = s_3 = n - 3$

     ……

     j = n-1, $s_j = s_{n-1} = n - (n-1) = 1$

Time Complexity of "*Sorting*" (2/3)

```
Sort-Array(A)
    for j = 1 to (A.length - 1)
        for i = (j + 1) to A.length
            if (A[i] < A[j])
                // swap A[i] and A[j]
                buffer = A[j]
                A[j] = A[i]
                A[i] = buffer
```

- Let us count the number f(n) of comparisons and express them as a function of n. This number f(n) is equal to the sum of:
  1. The outer loop (for j =1 ..) performs **n** comparisons (recall than A.length = n)
  2. The inner loop (for I = (j+1) ..) performs t_j comparisons. The number t_j depends on the value j:

     j = 1, $t_j = t_1 = n$
     j = 2, $t_j = t_2 = n - 1$
     j = 3, $t_j = t_3 = n - 2$
     ……
     j = n-1, $t_j = t_{n-1} = n - (n-1-1) = 2$

     $$\sum_{j=1}^{n-1} t_j = \sum_{k=2}^{n} k = \sum_{k=1}^{n} k - 1$$

  3. The body of the inner loop performs s_j comparisons (A[i] < A[j]).

     j = 1, $s_j = s_1 = n - 1$
     j = 2, $s_j = s_2 = n - 2$
     j = 3, $s_j = s_3 = n - 3$
     ……
     j = n-1, $s_j = s_{n-1} = n - (n-1) = 1$

Time Complexity of "*Sorting*" (2/3)

```
Sort-Array(A)
    for j = 1 to (A.length - 1)
        for i = (j + 1) to A.length
            if (A[i] < A[j])
                // swap A[i] and A[j]
                buffer = A[j]
                A[j] = A[i]
                A[i] = buffer
```

- Let us count the number f(n) of comparisons and express them as a function of n. This number f(n) is equal to the sum of:
  1. The outer loop (for j =1 ..) performs **n** comparisons (recall than A.length = n)
  2. The inner loop (for I = (j+1) ..) performs t_j comparisons. The number t_j depends on the value j:

     j = 1, $t_j = t_1 = n$
     j = 2, $t_j = t_2 = n - 1$
     j = 3, $t_j = t_3 = n - 2$
     ……
     j = n-1, $t_j = t_{n-1} = n - (n-1-1) = 2$

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

  3. The body of the inner loop performs s_j comparisons (A[i] < A[j]).

     j = 1, $s_j = s_1 = n - 1$
     j = 2, $s_j = s_2 = n - 2$
     j = 3, $s_j = s_3 = n - 3$
     ……
     j = n-1, $s_j = s_{n-1} = n - (n-1) = 1$

Time Complexity of "*Sorting*" (2/3)

```
Sort-Array(A)
    for j = 1 to (A.length - 1)
        for i = (j + 1) to A.length
            if (A[i] < A[j])
                // swap A[i] and A[j]
                buffer = A[j]
                A[j] = A[i]
                A[i] = buffer
```

t_j

s_j

- Let us count the number f(n) of comparisons and express them as a function of n. This number f(n) is equal to the sum of:
  1. The outer loop (for j =1 ..) performs **n** comparisons (recall than A.length = n)
  2. The inner loop (for I = (j+1) ..) performs t_j comparisons. The number t_j depends on the value j:

     j = 1, $t_j = t_1 = n$

     j = 2, $t_j = t_2 = n - 1$

     j = 3, $t_j = t_3 = n - 2$

     .....

     j = n-1, $t_j = t_{n-1} = n - (n-1-1) = 2$

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

  3. The body of the inner loop performs s_j comparisons (A[i] < A[j]).

     j = 1, $s_j = s_1 = n - 1$

     j = 2, $s_j = s_2 = n - 2$

     j = 3, $s_j = s_3 = n - 3$

     .....

     j = n-1, $s_j = s_{n-1} = n - (n-1) = 1$

Time Complexity of "*Sorting*" (2/3)

```
Sort-Array(A)
    for j = 1 to (A.length - 1)
        for i = (j + 1) to A.length
            if (A[i] < A[j])
                // swap A[i] and A[j]
                buffer = A[j]
                A[j] = A[i]
                A[i] = buffer
```

- Let us count the number f(n) of comparisons and express them as a function of n. This number f(n) is equal to the sum of:
  1. The outer loop (for j =1 ..) performs **n** comparisons (recall than A.length = n)
  2. The inner loop (for I = (j+1) ..) performs t_j comparisons. The number t_j depends on the value j:

     j = 1, $t_j = t_1 = n$
     j = 2, $t_j = t_2 = n - 1$
     j = 3, $t_j = t_3 = n - 2$
     ……
     j = n-1, $t_j = t_{n-1} = n - (n-1-1) = 2$

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

  3. The body of the inner loop performs s_j comparisons (A[i] < A[j]).

     j = 1, $s_j = s_1 = n - 1$
     j = 2, $s_j = s_2 = n - 2$
     j = 3, $s_j = s_3 = n - 3$
     ……
     j = n-1, $s_j = s_{n-1} = n - (n-1) = 1$

     $$\sum_{j=1}^{n-1} s_j$$

Time Complexity of "*Sorting*" (2/3)

```
Sort-Array(A)
    for j = 1 to (A.length - 1)
        for i = (j + 1) to A.length
            if (A[i] < A[j])
                // swap A[i] and A[j]
                buffer = A[j]
                A[j] = A[i]
                A[i] = buffer
```

t_j

s_j

- Let us count the number f(n) of comparisons and express them as a function of n. This number f(n) is equal to the sum of:
  1. The outer loop (for j =1 ..) performs **n** comparisons (recall than A.length = n)
  2. The inner loop (for I = (j+1) ..) performs t_j comparisons. The number t_j depends on the value j:

     j = 1, $t_j = t_1 = n$
     j = 2, $t_j = t_2 = n - 1$
     j = 3, $t_j = t_3 = n - 2$
     ……
     j = n-1, $t_j = t_{n-1} = n - (n-1-1) = 2$

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

  3. The body of the inner loop performs s_j comparisons (A[i] < A[j]).

     j = 1, $s_j = s_1 = n - 1$
     j = 2, $s_j = s_2 = n - 2$
     j = 3, $s_j = s_3 = n - 3$
     ……
     j = n-1, $s_j = s_{n-1} = n - (n-1) = 1$

     $$\sum_{j=1}^{n-1} s_j = \sum_{k=1}^{n-1} k$$

Time Complexity of "*Sorting*" (2/3)

```
Sort-Array(A)
    for j = 1 to (A.length - 1)
        for i = (j + 1) to A.length
            if (A[i] < A[j])
                // swap A[i] and A[j]
                buffer = A[j]
                A[j] = A[i]
                A[i] = buffer
```

t_j

s_j

- Let us count the number f(n) of comparisons and express them as a function of n. This number f(n) is equal to the sum of:
  1. The outer loop (for j =1 ..) performs **n** comparisons (recall than A.length = n)
  2. The inner loop (for I = (j+1) ..) performs t_j comparisons. The number t_j depends on the value j:

     j = 1, $t_j = t_1 = n$
     j = 2, $t_j = t_2 = n - 1$
     j = 3, $t_j = t_3 = n - 2$
     ……
     j = n-1, $t_j = t_{n-1} = n - (n-1-1) = 2$

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

  3. The body of the inner loop performs s_j comparisons (A[i] < A[j]).

     j = 1, $s_j = s_1 = n - 1$
     j = 2, $s_j = s_2 = n - 2$
     j = 3, $s_j = s_3 = n - 3$
     ……
     j = n-1, $s_j = s_{n-1} = n - (n-1) = 1$

     $$\sum_{j=1}^{n-1} s_j = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2}$$

Time Complexity of "*Sorting*" (2/3)

```
Sort-Array(A)
    for j = 1 to (A.length - 1)
        for i = (j + 1) to A.length
            if (A[i] < A[j])
                // swap A[i] and A[j]
                buffer = A[j]
                A[j] = A[i]
                A[i] = buffer
```

t_j

s_j

- Let us count the number f(n) of comparisons and express them as a function of n. This number f(n) is equal to the sum of:
  1. The outer loop (for j =1 ..) performs **n** comparisons (recall than A.length = n)
  2. The inner loop (for I = (j+1) ..) performs t_j comparisons. The number t_j depends on the value j:

     j = 1, $t_j = t_1 = n$
     j = 2, $t_j = t_2 = n - 1$
     j = 3, $t_j = t_3 = n - 2$
     ……
     j = n-1, $t_j = t_{n-1} = n - (n-1-1) = 2$

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

  3. The body of the inner loop performs s_j comparisons (A[i] < A[j]).

     j = 1, $s_j = s_1 = n - 1$
     j = 2, $s_j = s_2 = n - 2$
     j = 3, $s_j = s_3 = n - 3$
     ……
     j = n-1, $s_j = s_{n-1} = n - (n-1) = 1$

     $$\sum_{j=1}^{n-1} s_j = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Time Complexity of "*Sorting*" (2/3)

```
Sort-Array(A)
    for j = 1 to (A.length - 1)
        for i = (j + 1) to A.length
            if (A[i] < A[j])
                // swap A[i] and A[j]
                buffer = A[j]
                A[j] = A[i]
                A[i] = buffer
```

t_j

s_j

- Let us count the number f(n) of comparisons and express them as a function of n. This number f(n) is equal to the sum of:
  1. The outer loop (for j =1 ..) performs **n** comparisons (recall than A.length = n)
  2. The inner loop (for I = (j+1) ..) performs t_j comparisons. The number t_j depends on the value j:

     j = 1, $t_j = t_1 = n$
     j = 2, $t_j = t_2 = n - 1$
     j = 3, $t_j = t_3 = n - 2$
     ……
     j = n-1, $t_j = t_{n-1} = n - (n-1-1) = 2$

     $$\sum_{j=1}^{n-1} t_j = \frac{1}{2}n^2 + \frac{1}{2}n - 1$$

  3. The body of the inner loop performs s_j comparisons (A[i] < A[j]).

     j = 1, $s_j = s_1 = n - 1$
     j = 2, $s_j = s_2 = n - 2$
     j = 3, $s_j = s_3 = n - 3$
     ……
     j = n-1, $s_j = s_{n-1} = n - (n-1) = 1$

     $$\sum_{j=1}^{n-1} s_j = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Time Complexity of "*Sorting*" (2/3)

```
Sort-Array(A)
    for j = 1 to (A.length - 1)
        for i = (j + 1) to A.length
            if (A[i] < A[j])
                // swap A[i] and A[j]
                buffer = A[j]
                A[j] = A[i]
                A[i] = buffer
```

t_j

s_j

- Let us count the number f(n) of comparisons and express them as a function of n. This number f(n) is equal to the sum of:
  1. The outer loop (for j =1 ..) performs **n** comparisons (recall than A.length = n)
  2. The inner loop (for I = (j+1) ..) performs t_j comparisons. The number t_j depends on the value j:

     j = 1, $t_j = t_1 = n$
     j = 2, $t_j = t_2 = n - 1$
     j = 3, $t_j = t_3 = n - 2$
     ……
     j = n-1, $t_j = t_{n-1} = n - (n-1-1) = 2$

     $$\sum_{j=1}^{n-1} t_j = \frac{1}{2}n^2 + \frac{1}{2}n - 1$$

  3. The body of the inner loop performs s_j comparisons (A[i] < A[j]).

     j = 1, $s_j = s_1 = n - 1$
     j = 2, $s_j = s_2 = n - 2$
     j = 3, $s_j = s_3 = n - 3$
     ……
     j = n-1, $s_j = s_{n-1} = n - (n-1) = 1$

     $$\sum_{j=1}^{n-1} s_j = \frac{1}{2}n^2 - \frac{1}{2}n$$

Time Complexity of "*Sorting*" (3/3) Summary

```
Sort-Array(A)
        for j = 1 to (A.length - 1)
                for i = (j + 1) to A.length
                        if (A[i] < A[j])
                                // swap A[i] and A[j]
                                buffer = A[j]
                                A[j] = A[i]
                                A[i] = buffer
```

s_j

- Let us **sum up** the number f(n)of comparisons and express them as a function of n. This number f(n) is equal to the **sum** of:
  1. **n**: number of comparisons performed by the **outer** loop (for j =1 ..).
  2. **$1/2n^2 + 1/2n - 1$**: number of comparisons performed by the **inner** loop (for i = (j+1) ..)
  3. **$1/2n^2 - 1/2n$**: number of comparisons performed in the body of the **inner** (A[i] < A[j]).
- **Conclusion**. The total number of comparisons is **$n^2 - 1$** …
- Let us reflect on this: **What if we included the swap operations in the count?**
- We would get something like **$a.n^2 + b.n + c$**. The coefficients a, b, and c are not critical because they are **independent from** the input size **n**: if ***n*** increases, ***a, b,*** and ***c*** do NOT!
- What really matters is that time complexity will "grow" as **n^2**. Can we justify this?

Space Complexity of "Sorting"

```
Sort-Array(A)
      for j = 1 to (A.length - 1)
            for i = (j + 1) to A.length
                  if (A[i] < A[j])
                        // swap A[i] and A[j]
                        buffer = A[j]
                        A[j] = A[i]
                        A[i] = buffer
```

- What is the unit of space?
  - Space for one item in the array or a variable (i, j, and buffer)
  - The size of the input is ***n*** (***n*** items in the array)
  - count and express this amount of space as a function of the *size of the input* ***n***:
    - The array requires n items → **n space units**
    - There are three variables: i, j, and buffer. → **3 space units**
  - In total, this algorithm requires **n + 3 space units**.
- Let us reflect on this: **What if we included the size to store the algorithm itself or use bytes as units?**
- We would get something like **a.n + b**. The constants ***a*** and ***b*** are not critical because they are **independent from** the input size **n**: if ***n*** increases, ***a*** and ***b*** do NOT!
- What really matters is that space complexity will "grow" as **n**. Can we justify this?

Upper Bound (Asymptotic Notation)

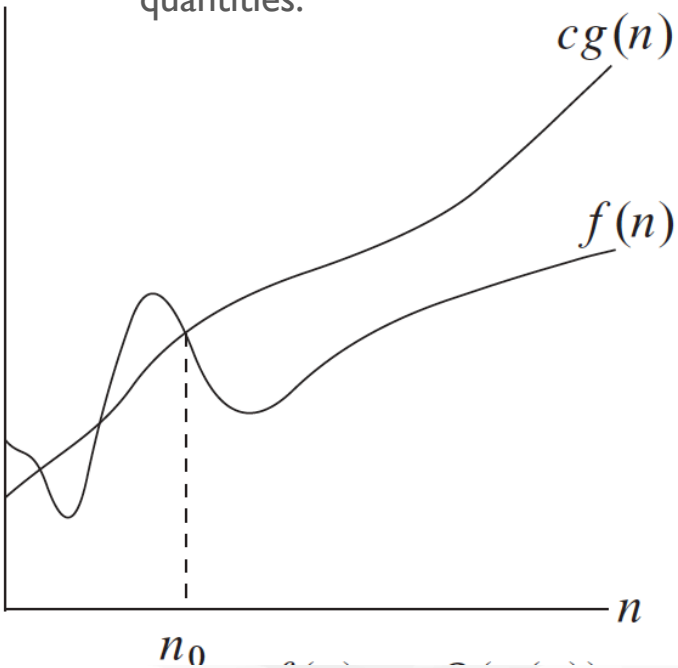
- What is an asymptotic **upper bound ?**
- Definition of the set **O**(g(n))

- Examples:
  - Example 1: f(n) = a.n + b
    - Show that f(n) belongs to the set **O**(g(n)) with g(n) = n.
    - In other words, show that f(n) belongs to **O**(n) (i.e., f(n) € O(n))
  - Example 2: f(n) = a.n^2 + b.n + 1
    - Show that f(n) belongs to the set **O**(g(n)) with g(n) = n^2.
    - In other words, show that f(n) belongs to **O**(n^2) (i.e., f(n) € O(n^2))

Read Section 3.1 (O notation)

Asymptotic Upper Bound

- Definition
  - $O(g(n))$ = {$f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le c\, g(n)$ for all $n \ge n_0$}
- Remarks
  - $O(g(n))$ is a **set** of functions
  - If a function belongs to the set , it means that the function $g(n)$ is an asymptotic upper bound to $f(n)$
  - We are considering only positive functions: time complexity and space complexity are positive quantities.



$$f(n) \in O(g(n))$$

Example 1: f(n) = a.n+b (a > 0) , g(n) = n

- Let us show: $a.n + b \in O(n)$
- To show that f(n) = a.n+b belongs to O(n), we need to show that:
  - there exist constants c and n_0 such that $0 \leq a.n+b \leq c.n$ for all $n \geq n_0$.
- We just need to find c and n_0 such that $0 \leq a.n + b \leq c.n$
- Dividing by n all terms, we get: $0 \leq a + \frac{b}{n} \leq c$
- Let us choose any $n_0 \geq 1$ and a number $c = c_0$ such that $0 \leq a + \frac{b}{n_0} \leq c_0$
- In this case we will have for all $n \geq n_0$ $0 \leq a + \frac{b}{n} \leq c_0$
- and therefore $0 \leq a.n + b \leq c_0.n$
- meaning that $a.n + b \in O(n)$

Example 1: f(n) = a.n+b (a > 0) , g(n) = n (Continued)

- We showed that $a.n + b \in O(n)$
- This means that any time complexity or space complexity that is expressed as a first degree polynomial belongs to O(n)
- In other words, the coefficients a and b are not important for the asymptotic behavior of the function a.n+b.
- By **ABUSE**, we write a.n + b **=** O(n) to mean $a.n + b \in O(n)$

- If a time complexity or space complexity is expressed as a first degree polynomial, it means that asymptotically, there is an **upper bound** function g(n) = c.n to that time/ space complexity.

Example 11: f(n) = a.n^2+b.n + d, g(n) = n^2

- Let us show: $a.n^2 + b.n + d \in O(n^2)$
- To show that f(n) = a.n^2+b.n+d belongs to O(n^2), we need to show that:
  - there exist constants c and n_0 such that $0 \leq a.n^2+b.n+d \leq c.n^2$ for all $n \geq n_0$.
- We just need to find c and n_0 such that $0 \leq a.n^2 + b.n + d \leq c.n^2$
- Dividing by n^2 all terms, we get: $0 \leq a + \frac{b}{n} + \frac{d}{n^2} \leq c$
- Let us choose any $n_0 \geq 1$ and a number $c = c_0$ such that $0 \leq a + \frac{b}{n_0} + \frac{d}{n_0^2} \leq c_0$
- In this case we will have for all $n \geq n_0$ $0 \leq a + \frac{b}{n} + \frac{d}{n^2} \leq c_0$
- and therefore $0 \leq a.n^2 + b.n + d \leq c_0.n^2$
- meaning that $a.n^2 + b.n + d \in O(n^2)$

Example 11: $f(n) = a.n^2+b.n + d,\ g(n) = n^2$ (Continued)

- We showed that $(a.n^2 + b.n + d) \in O(n^2)$
- This means that any time complexity or space complexity that is expressed as a second degree polynomial belongs to $O(n^2)$
- In other words, the coefficient a,b, and d are not important for the asymptotic behavior of the function $f(n) = a.n^2+b.n+d$.
- By **ABUSE**, we write $a.n^2+b.n+d$ **=** $O(n^2)$ to mean $(a.n^2 + b.n + d) \in O(n^2)$

- If a time complexity or space complexity is expressed as a second degree polynomial, it means that asymptotically, there is an **upper bound** function $g(n) = c.n^2$ to that time/space complexity.

Lower Bound (Asymptotic Notation)

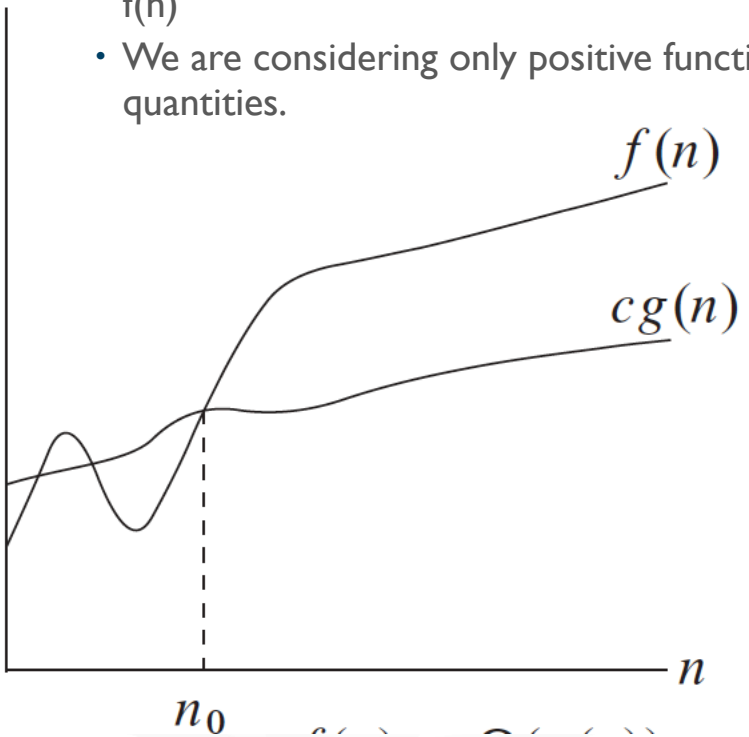
- What is an asymptotic **lower bound ?**
- Definition of the set **Ω**(g(n))

- Examples:
  - Example 1: $f(n) = a.n + b$
    - Show that f(n) belongs to the set Ω(g(n)) with $g(n) = n$.
    - In other words, show that f(n) belongs to Ω(n) (i.e., $f(n) \in \Omega(n)$)
  - Example 2: $f(n) = a.n^2 + b.n + 1$
    - Show that f(n) belongs to the set Ω(g(n)) with $g(n) = n^2$.
    - In other words, show that f(n) belongs to $\Omega(n^2)$ (i.e., $f(n) \in \Omega(n^2)$)

Read Section 3.1 (Ω notation)

Asymptotic Lower Bound

- Definition
  - $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le c\, g(n) \le f(n)$ for all $n \ge n_0\}$
- Remarks
  - $\Omega(g(n))$ is a **set** of functions
  - If a function belongs to the set , it means that the function g(n) is an asymptotic **lower** bound to f(n)
  - We are considering only positive functions: time complexity and space complexity are positive quantities.



$$f(n) \in \Omega(g(n))$$

Example 1: f(n) = a.n+b (a > 0, b>0), g(n) = n

- Let us show: $a.n + b \in \Omega(n)$
- To show that f(n) = a.n+b belongs to Ω (n), we need to show that:
  - there exist constants c and n_0 such that $0 \le c.n \le a.n+b$ for all $n \ge n_0$.
- We just need to find c and n_0 such that $0 \le c.n \le a.n + b$
- Dividing by n all terms, we get: $0 \le c \le a + \frac{b}{n}$
- Let us choose any n_0 and a number $c = c_0$ such that $0 \le c \le a$
- In this case we will have for all $n \ge n_0$ $0 \le c_0 \le a + \frac{b}{n}$
- and therefore $0 \le c_0 n \le a.n + b$
- meaning that $a.n + b \in \Omega(n)$

Example 1: f(n) = a.n+b (a > 0), g(n) = n (Continued)

- We showed that $a.n + b \in \Omega(n)$
- This means that any time complexity or space complexity that is expressed as a first degree polynomial belongs to Ω(n)
- In other words, the coefficient a and b are not important for the asymptotic behavior of the function a.n+b.
- By **ABUSE**, we write a.n + b **=** Ω(n) to mean $a.n + b \in \Omega(n)$

- If a time complexity or space complexity is expressed as a first degree polynomial, it means that asymptotically, there is an **lower bound** function g(n) = c.n to that time/ space complexity.

Example 11: f(n) = a.n^2+b.n + d(a,b, and d positive), g(n) = n^2

- Let us show: $a.n^2 + b.n + d \in \Omega(n^2)$
- To show that f(n) = a.n^2+b.n+d belongs to $\Omega(n^2)$, we need to show that:
  - there exist constants c and n_0 such that $0 \leq c.n^2 \leq a.n^2+b.n+d$ for all $n \geq n_0$.
- We just need to find c and n_0 such that $0 \leq c.n^2 \leq a.n^2 + b.n + d$
- Dividing by n^2 all terms, we get: $0 \leq c \leq a + \frac{b}{n} + \frac{d}{n^2}$
- Let us choose n_0 and a number c = c_0 such that $0 \leq c_0 \leq a + \frac{b}{n} + \frac{d}{n^2}$
- In this case, let us simply choose c0 such that $0 \leq c_0 \leq a$
- and therefore $0 \leq c_0.n^2 \leq a.n^2 + b.n + d$
- meaning that $a.n^2 + b.n + d \in \Omega(n^2)$

Example 11: $f(n) = a.n^2+b.n + d,\ g(n) = n^2$ (Continued)

- We showed that $(a.n^2 + b.n + d) \in \Omega(n^2)$
- This means that any time complexity or space complexity that is expressed as a second degree polynomial belongs to $\Omega(n^2)$
- In other words, the coefficient a,b, and d are not important for the asymptotic behavior of the function $f(n) = a.n^2+b.n+d$.
- By **ABUSE**, we write $a.n^2+b.n+d$ **=** $\Omega(n^2)$ to mean $(a.n^2 + b.n + d) \in \Omega(n^2)$

- If a time complexity or space complexity is expressed as a second degree polynomial, it means that asymptotically, there is an **lower bound** function $g(n) = c.n^2$ to that time/space complexity.

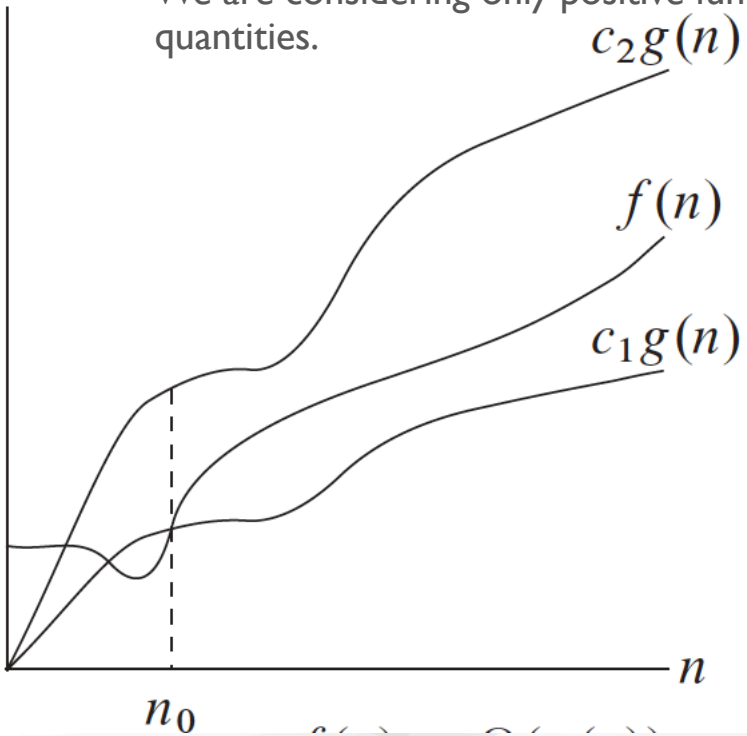
Tight Bound (Asymptotic Notation)

- What is an asymptotic **tight bound ?**
- Definition of the set $\Theta(g(n))$
- Relationship between $O(g(n))$, $\Omega(g(n))$, and $\Theta(g(n))$
- Generalization to an m degree polynomial $P_m(n)$

- Read Section 3.1 (Θ notation)

Asymptotic Tight Bound

- Definition
  - $\Theta(g(n))$ = {f(n): there exist positive constants c_1, c_2, and n_0 such that
    $0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)$ for all $n \ge n_0$}
- Remarks
  - $\Theta(g(n))$ is a **set** of functions
  - If a function belongs to the set $\Theta(g(n))$, it means that the function g(n) is an asymptotic **tight** bound to f(n)
  - We are considering only positive functions: time complexity and space complexity are positive quantities.



$$f(n) \in \Theta(g(n))$$

Asymptotic Tight Bound

- **Theorem**
  - A function $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$
  - In words, this theorem says that a function g(n) is asymptotically a tight bound to f(n) if and only if g(n) is an upper bound and a lower bound to f(n).
- **Remarks**
  - We showed that $f(n) = a.n+b$ belongs to $O(n)$ and $\Omega(n)$. Therefore, $(a.n+b) \in \Theta(n)$
  - We showed that $f(n) = a.n^2+b.n+c$ belongs to $O(n^2)$ and $\Omega(n^2)$. Therefore, $(a.n^2+b.n+c) \in \Theta(n^2)$
  - By abuse, we write $f(n) = \Theta(g(n))$ to mean $f(n) \in \Theta(g(n))$
  - If f(n) is an m degree polynomial P_m ($f(n) = P_m = a_m.n^m + a_{m-1}.n^{m-1} + \ldots + a_1.n + a_0$), then

$$P_m \in \Theta(n^m)$$

  - Abusing this notation, we write $Pm = \Theta(n^m)$
- **Conclusion**:
  - Time or space complexity of algorithms will be characterized by their asymptotic upper, lower, or tight bound.
  - For example:
    - The **time** complexity of the naïve sorting algorithm is $\Theta(n^2)$
    - The **space** complexity of the naïve sorting algorithm is $\Theta(n)$

Wrap Up

- Learn and understand how to characterize the efficiency f(n) of algorithms.
- Learn, understand, and use the asymptotic notation.
  - Find for f(n) an Upper bound g(n) such that f(n) € O(g(n))
  - Find for f(n) a lower bound g(n) such that f(n) € Ω(g(n))
  - Find for f(n) a tight bound g(n) such that f(n) € (Θ(g(n))
- Learn, understand, and use growth of functions to characterize efficiency.
  - An m degree polynomial P_m € Θ(n^m)
- How to best use these bounds?
  - Think about when you ask a friend how much you will pay for a meal at a restaurant. The friend will say:
    - minimum (lower bound) is $1 and maximum (upper bound) is $10,000
    - lower bound is $5 and upper bound is $200
    - lower bound is $18 and upper bound is $25
    - tight bound : about $20
    - Conclusion: you prefer the **largest** lower bound and the **smallest** upper bound. Best is when both bounds are equal → tight bound
  - Let us apply the above for a function f(n).
    - lower bound: find the *largest* $g_1(n)$ such that f(n) € Ω($g_1(n)$)
    - upper bound: find the *smallest* $g_2(n)$ such that f(n) € O($g_2(n)$)
    - tight bound: g(n) = $g_1(n)$ = $g_2(n)$ such that f(n) € Θ(g(n))