Module 1

Introduction and Complexity Analysis

Data Structure

- A data structure is a format for **data organization, management,** and **storage** to enable **efficient access** and **modification**

OR

- A logical or mathematical model to organize, manage, and store the data efficiently

Abstract Data Type (ADT)

Types of Data Structures

- **Linear Data Structures:** Data elements form a sequence or a linear list. The data is arranged in a linear fashion although the way they are stored in the memory need not to be sequential
  - Array
  - Linked List
  - Stack
  - Queue
- **Non-linear Data Structures:** Data elements is not arranged in sequence
  - Tree
  - Graph

Example

Integer Array of Size 100

vs.

100 Integer Variables

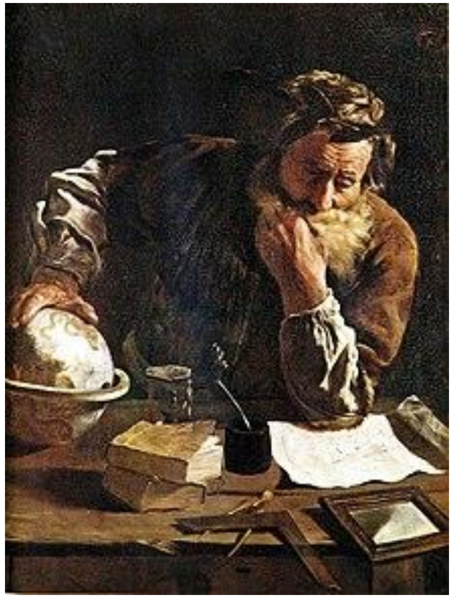
- Efficiently locate, retrieve, update information
- Understand logical relationship among data elements

Story Time!

Once Upon a Time in a Land Far Away

There Lived a Mathematician

Archimedes



Algorithms

- What are Algorithms and why it is important to study about them?

An algorithm is any **well-defined computational procedure** that takes some value(s), as **input** and produces some value(s), as **output**

OR

An algorithm is a **sequence of computational steps** that transform the **input** into the **output**

Hence,

An algorithm can be viewed as a **tool** for solving a well-defined ***Computational Problem***

Algorithm vs. Program

- An algorithms is an abstract computation procedure,

Can be expressed in many ways

while a program is an expression of an algorithm

- A program follows a strict syntax, while an algorithm can be written in a human-understandable high level language

Pseudocode

*An algorithm is a step by step procedure to solve a given problem while a pseudocode is a **method** of writing an algorithm*

Pseudocode Conventions

- Indentation indicates block structure: for, while, if-else
  - May use { } occasionally
  - **begin** and **end** statements are not used
- **while**, **for**, **repeat**-**until** and **if**-**else** conditional construct have interpretations similar to C, C++, Java, and Python
  - The loop counter retains its value after exiting the loop
  - The loop counter’s value is the value that first exceeded the **for** loop bound
- The symbol “**//**” indicates a comment
- Variables are local to the given procedure
  - Global variables are explicitly declared

Pseudocode Conventions

- `A[i]` indicates the i^{th} element of the array A
  - Array index starts with 1 (not 0 as in C, C++, and Java)
  - `A[1..j]` indicates the subarray of A i.e., `A[1], A[2],…,A[j]`
- By default, we pass parameters to a procedure ***by value***
- A **return** statement immediately transfers control back to the point of call in the calling procedure
- The Boolean operators “and” and “or” are ***short circuiting***
- The keyword **error** indicates that an error occurred
- ‘==’ for equality and ‘=’ for assignment

Problem: Linear Search

- **Input:** A sequence of n numbers A[1..n] and a value *v*
- **Output:** Index `i` such that `v==A[i]` or NULL if value does not appear in A
- Pseudocode

```
SEARCH(A, v)
  for i = 1 to A.length
      if A[i] == v
          return i
  return NULL
```

i=6

- Input sequence (**instance**): <5,2,6,8,9,4,3,2>, Value v = 4

Problem: Finding Largest Element

- **Input:** A sequence of n numbers A[1..n]
- **Output:** Largest element in A
- Pseudocode

```
SEARCH-LARGE(A)
large = A[1]
for i = 2 to A.length
    if A[i] > large
        large = A[i]
return large
```

Largest Element

- Input sequence (instance): <5,2,6,8,9,4,3,2>

Problem: Sum of Elements

- **Input:** A sequence of n numbers A[1..n]
- **Output:** Sum of elements of A
- Pseudocode

```
SUM(A)
sum = 0
for i = 1 to A.length
    sum = sum + A[i]
return sum
```

- Input sequence (instance): <5,2,6,8,9,4,3,2>
- Output: 39

Design and Analysis

- Design and Analysis
  - Algorithm should be **Correct**: For **every** input instance, it halts with the correct output
    - Incorrect algorithms: Useful sometimes, if we can control their error rate
  - Algorithm should be **Efficient**
    - **Running/Execution Time: Time Complexity**
    - Space Requirement: Space Complexity
    - Other Factors:
      - Network (web or cloud based application)
      - Power consumption (laptop/tablet/pc/mobile)
      - CPU registers

Priori and Posteriori Analysis

- Running time depends on
  - Single vs Multi processor
  - Read or Write speed to Memory
  - 16 bit vs 32 bit vs 64 bit
  - Input size and type

YES/NO??

PRIORI ANALYSIS	POSTERIORI ANALYSIS
Priori analysis is an absolute analysis.	Posteriori analysis is a relative analysis.
It is independent of language of compiler and types of hardware.	It is dependent on language of compiler and type of hardware.
It will give approximate answer.	It will give exact answer.

Time and Space Analysis

- **Example 1:** Swap two numbers

- **Input:** Two numbers **a** and **b** to be swapped
- **Output:** Values of **a** and **b** swapped

```
SWAP(a,b)
    temp = a
    a=b
    b=temp
```

	Time*
	1 Unit
	1 Unit
	1 Unit
Total	3 Units

Space*	
a:	1 Unit
b:	1 Unit
temp:	1 Unit
Total	3 Units

Frequency Count Method

*RAM Model: Algorithm Design By Michael T. Goodrich

Frequency Count Method

Example 2: Sum of the numbers of an array

- **Input:** An array **A[1..n]** of size **n**
- **Output:** Sum of elements of **A**

```
SUM(A)
S = 0
for i = 1 to A.length
      S = S + A[i]
```

Frequency Count Method

Example 3: Sum of two n-dimensional matrices

- **Input:** Two array **A** and **B** of size **nxn**
- **Output:** Sum of matrices stored in matrix **C**

```
SUM(A, B)
   for i = 1 to A.length
       for j = 1 to B.length
           C[i][j] = A[i][j] + B[i][j]
```

Frequency Count Method

Example 3: Multiplication of two n-dimensional matrices

- **Input:** Two array **A** and **B** of size **nxn**
- **Output:** Multiplication of matrices stored in matrix **C**

```
SUM(A, B)
for i = 1 to A.length
    for j = 1 to B.length
        C[i][j] = 0
        for k = 1 to A.length
            C[i][j] = C[i][j] + A[i][k] × B[k][j]
```

Exercises

- Calculate Time Complexity: Degree of the Polynomial

```
for (i=1; i <= n; i++)
        a statement

for (i=n; i >=1; i--)
        a statement

for (i=1; i <= n; i++)
        for (j=1; j <= n; j++)
                a statement

for (i=1; i <= n; i++)
        for (j=1; j <= i; j++)
                a statement
```

Exercises

```
x = 0
for (i=1; x ≤ n; i++)
        x = x + i

for (i=1; i < n; i = i×2)
        a statement

for (i=n; i > 1; i = i/2)
        a statement
```

Exercises

```
for (i=0; i < n; i++)
        a statement
for (j=0; j < n; j++)
        a statement
```

```
x = 0
for (i=1; i ≤ n; i = i×2)
        x = x+1
for (j=1; j ≤ x; j = j×2)
        a statement
```

```
for (i=0; i < n; i++)
        for (j=1; j < n; j = j×2)
                a statement
```

Classes of Functions

- f (n) = 2 Constant
- f (n) = 200 Constant
- f (n) = 2000 Constant
- f (n) = log n Logarithmic
- f (n) = n Linear
- f (n) = n^2 Quadratic
- f (n) = n^3 Cubic
- f (n) = 2^n Exponential

Compare classes of functions

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < n^n$$

$Input\ Size(n)$	$\log n$	n	$nlogn$	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	10^3	10^4	10^6	10^9	10^{300}

Asymptotic Notations

- Mathematical notations to represent the time function (complexity) of an algorithm.
- Used to define the **growth rate of an algorithm** as the input size is increased.
- Performance of an algorithm in-terms of the input size
- Three standard asymptotic notations:
  - Big-Oh O →upper bound
  - Big-Omega Ω →lower bound
  - Theta θ →lower and upper bound

Big-Oh O

- Definition

The function $f(n) = O\big(g(n)\big)$

there exists **positive** constants c > 0 and n_0 such that $f(n) \leq c\ g(n)$

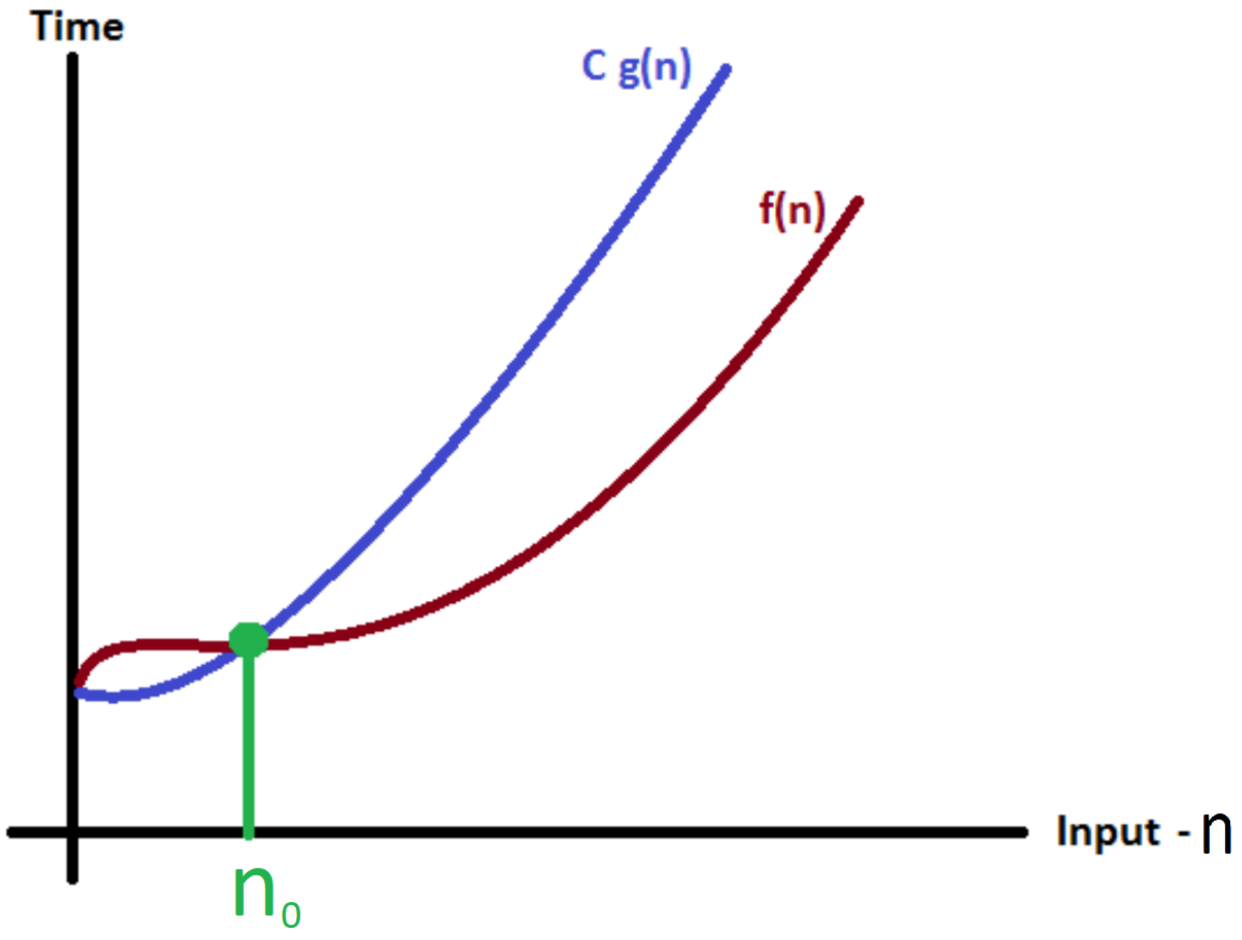
$\forall n \geq n_0$ where n_0 ≥ 1

Example

$$f\ (n) = 3n + 2$$

Select a value of c and n_0 such that f (n) is always lesser than or equal to the g(n)

Graphical Representation Big-Oh O



Big-Omega Ω

- Definition

The function $f(n) = \Omega(g(n))$

there exists **positive** constants c > 0 and n_0 such that $f(n) \geq c\ g(n)$

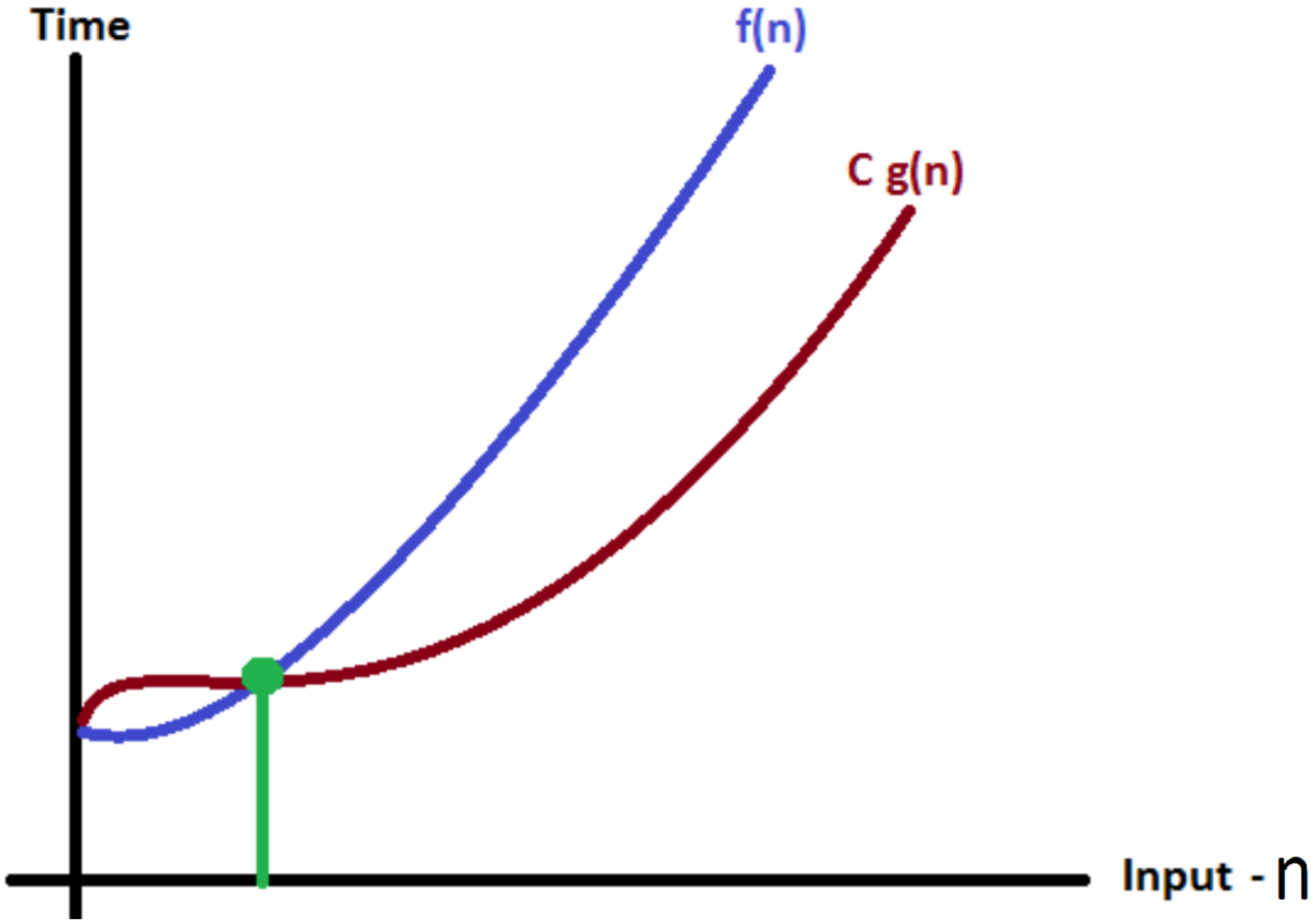
$\forall n \geq n_0$ where n_0 ≥ 1

Example

$$f\ (n)\ =\ 3n\ +\ 2$$

Select a value of c and n_0 such that f (n) is always lesser than or equal to the g(n)

Graphical Representation Big-Omega Ω



Big-Theta Θ

- Definition

The function $f\ (n)\ = \Theta(g(n))$
there exists constants $c_1, c_2 > 0$ and n_0 such that $c_1\ g(n)\ \leq f(n) \leq\ c_2\ g(n)$
$\forall n\ \geq\ n_0$ where $n_0 \geq 1$

- Primarily used when both upper bound and lower bound functions are equal
- Example

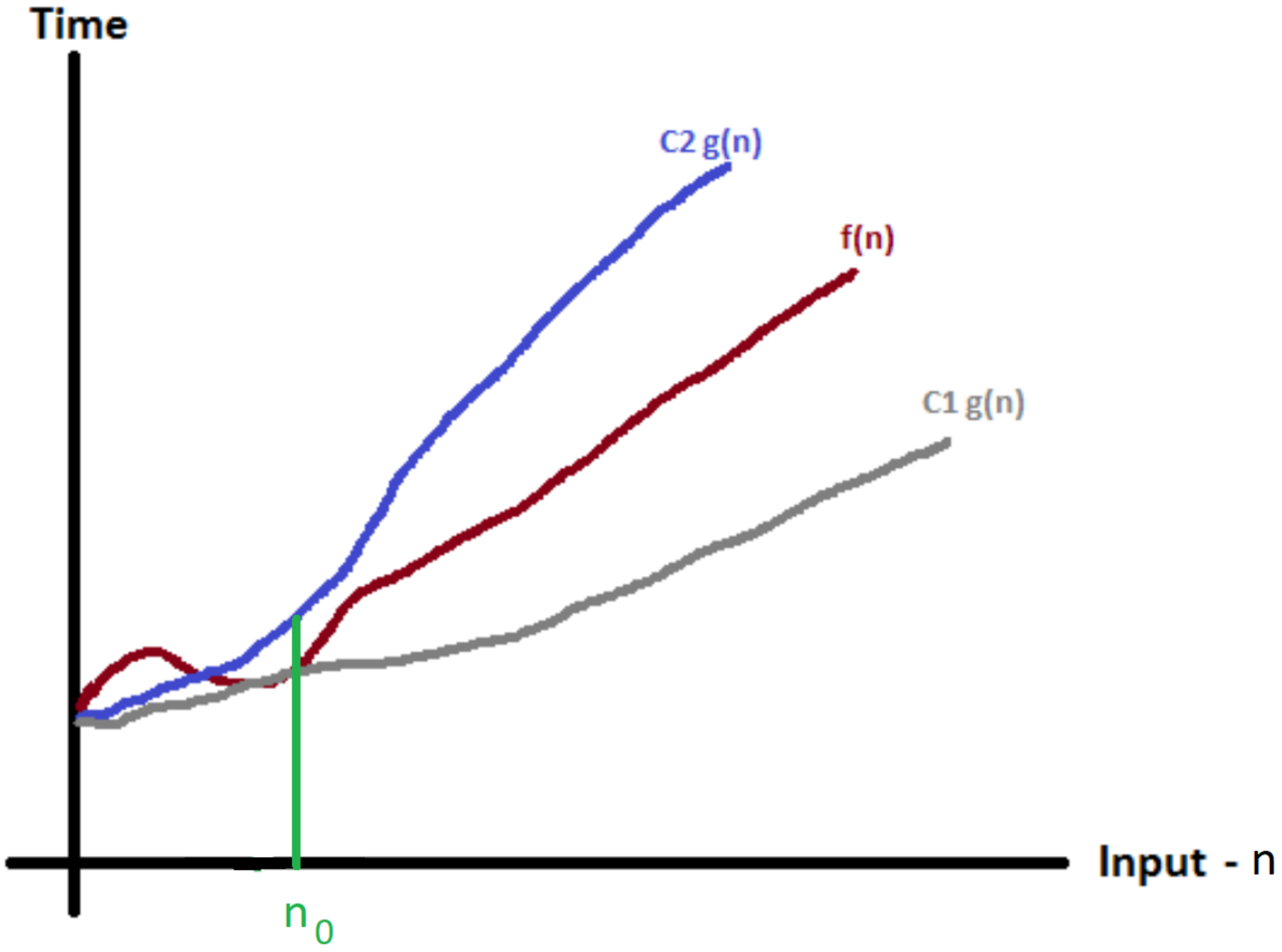
$$f\ (n)\ =\ 2n\ +\ 3$$

$g(n)\ =\ n$ for lower bound and $g(n)\ =\ 5n$ for upper bound $\forall n\ \geq\ 1$

$$n \leq 2n\ +\ 3 \leq 5n$$

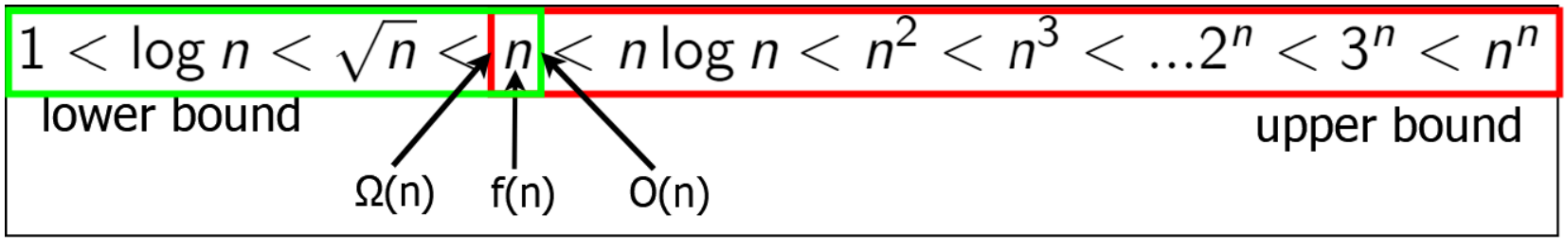
$$f\ (n)\ =\ \Theta(g(n))$$

Graphical Representation Big-Theta Θ



Recap

- While computing the time function of any algorithm, always select the highest degree of polynomial
    - constant 1
    - linear n
    - quadratic n^2
    - cubic n^3
    - exponential 2^n and n^n
- While computing the bound, always select the closest value



Example: Array Search

- Given an array [5, 4, 3, 2, 1, 9, 8, 7, 6] Search for element x

```
SEARCH(A, n, v)
 for i = 1 to n
     if A[i] == v
         return i
  return NULL
```

- Lower Bound $\Omega(1)$ BEST Case
- Upper Bound O(n) WORST Case

Will it be $\Theta(n)$?

Case vs. Bound

Average Case

- Element to search (x) is equally likely to occur any position in the array → x can occur at any array index with probability 1/n

$$f(n) = 1.\frac{1}{n} + 2.\frac{1}{n} + 3.\frac{1}{n} + \cdots + n.\frac{1}{n}$$

$$= \frac{n(n+1)}{2}.\frac{1}{\mathrm{n}} = \frac{\mathrm{n}+1}{2}$$

Hence,

$$f(n) = \Theta(\mathrm{n})$$

Is this better than the worst case?

Tricky Stuff!!

- Algo 1: $f(n) = \log_2 n + 3$
- Algo 2: $f(n) = 5\log_6 n + 1$

- $f(n) = n^2 - 8n$ $f(n) = \Omega(n^2)$

- Comparing two functions

$f(n) = n^2 + 8n$ $g(n) = 4n^3 + 2$

$f(n) = O\big(g(n)\big)$?

$f(n) = \Omega\big(g(n)\big)$?

$f(n) = \Theta\big(g(n)\big)$?

Little-oh o

- Definition

The function $f(n) = o\big(g(n)\big)$

For all **positive** constants c > 0, there exists a constant n_0 such that $f(n) < c\, g(n)\ \forall n \geq n_0$ where $n_0 \geq 1$

- Examples

$f\ (n)\ =\ 3n\ +\ 2$ $\qquad f(n) = o(n^2)$

$f\ (n)\ = 4n^3 + 5$ $\qquad f(n) = o(n^4)$

Little-oh o

- Intuitively, in o-notation, the function f(n) becomes insignificant relative to g(n) as n approaches infinity

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Little-oh: An upper bound that is not asymptotically tight

Little-omega ω

- Definition

The function $f(n) = \omega\big(g(n)\big)$

For all **positive** constants c > 0, there exists a constant n_0 such that $f(n) > c\ g(n)\ \forall n \geq n_0$ where $n_0 \geq 1$

- Examples

$f\ (n)\ =\ 3n\ +\ 2$ $\qquad f(n) = \omega(1)$

$f\ (n)\ = 4n^3 + 5$ $\qquad f(n) = \omega(n^2)$

Little-omega ω

- Intuitively, in ω-notation, the function f(n) becomes arbitrarily large relative to g(n) as n approaches infinity

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Little-omega: A lower bound that is not asymptotically tight

Comparing functions

- Transitivity:

$$
\begin{aligned}
f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) &\quad \text{imply} \quad f(n) = \Theta(h(n)) \\
f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) &\quad \text{imply} \quad f(n) = O(h(n)) \\
f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) &\quad \text{imply} \quad f(n) = \Omega(h(n)) \\
f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) &\quad \text{imply} \quad f(n) = o(h(n)) \\
f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) &\quad \text{imply} \quad f(n) = \omega(h(n))
\end{aligned}
$$

- Reflexivity:

$$
\begin{aligned}
f(n) &= \Theta(f(n)) \\
f(n) &= O(f(n)) \\
f(n) &= \Omega(f(n))
\end{aligned}
$$

Comparing functions

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

Transpose symmetry:

$$f(n) = O(g(n)) \quad \text{if and only if} \quad g(n) = \Omega(f(n))$$

$$f(n) = o(g(n)) \quad \text{if and only if} \quad g(n) = \omega(f(n))$$

Comparing functions

- Comparing asymptotic complexities of two functions as two real numbers a and b

$$
\begin{array}{lll}
f(n) = O(g(n)) & \text{is like} & a \le b \\
f(n) = \Omega(g(n)) & \text{is like} & a \ge b \\
f(n) = \Theta(g(n)) & \text{is like} & a = b \\
f(n) = o(g(n)) & \text{is like} & a < b \\
f(n) = \omega(g(n)) & \text{is like} & a > b
\end{array}
$$

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, **"Introduction to Algorithms"**, The MIT Press
- Sahni, S., **"Data Structures, Algorithms, and Applications in C++",** WCB/McGraw-Hill
- Algorithms, Video Lectures by Abdul Bari, 1.1-1.12

https://www.youtube.com/playlist?list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O