Algorithm Analysis

IT5003: Data Structures and Algorithms
(**AY2019/20 Semester 1**)

Lecture Outline

- What is an Algorithm?
- What is Analysis of Algorithms?

- How to analyze an algorithm
- Big-O notation
  - Big-Omega & Big-Theta

- Example Analyses

You are expected to know…

- Proof by induction
- Operations on logarithm function
- Arithmetic and geometric progressions
  - Their sums
- Linear, quadratic, cubic, polynomial functions
- Ceiling, floor, absolute value

Algorithm and Analysis

- **Algorithm**
  - A step-by-step procedure for solving a problem

- **Analysis of Algorithm**
  - To evaluate rigorously the **resources** (**time** and **space**) needed by an algorithm and **represent the result of the evaluation with a formula**
  - We focus more on **time** requirement in our analysis
  - The time requirement of an algorithm is also called the **time complexity** of the algorithm

Measure Actual Running Time?

- We can measure the actual running time of a **program**
  - Use **wall clock time** or **insert timing code** into program

- However, running time is not meaningful when **comparing two algorithms:**
  a. Coded in different languages
  b. Using different data sets
  c. Running on different computers

Counting Operations

- Instead, we count the number of **operations**
  - e.g. *arithmetic*, *assignment*, *comparison*, etc.

- Counting an algorithm's operations is a way to assess its **efficiency**
  - An algorithm's execution time is related to the number of operations it requires

Example: Counting Operations

```
for i in range(1, n+1):
    perform 100 operations;          // A
    for j in range(0, n+1):
        perform 2 operations;        // B
```

Total Ops = **A + B** $= \sum_{i=1}^{n} 100 + \sum_{i=1}^{n} (\sum_{j=1}^{n} 2)$

$$= 100n + \sum_{i=1}^{n} 2n \quad = 100n + 2n^2 \quad = 2n^2 + 100n$$

Example: **Counting Operations**

- Knowing the number of operations required by the algorithm, we can state that
  - **Algorithm *X*** takes $2n^2 + 100n$ operations to solve problem of size n
- If the time ***t*** needed for one operation is known, then we can state
  - **Algorithm *X*** takes $(2n^2 + 100n)\ t$ time units

Example: **Counting Operations**

- However, time ***t*** is directly dependent on the factors mentioned earlier
  - E.g. different languages, compilers and computers

- Instead of tying the analysis to actual time ***t***, we can state
  - ***Algorithm X*** takes time that is **proportional to** $2n^2 + 100n$ for solving problem of size n

Approximation of Analysis Results

- Suppose the time complexity of
  - Algorithm ***A*** is $3n^2 + 2n + \log n + 30$
  - Algorithm ***B*** is $0.39n^3 + n$
- Intuitively, we know Algorithm *A* will outperform *B*
  - When solving larger problem, i.e. larger ***n***
- The **dominating term** $3n^2$ and $0.39n^3$ can tell us approximately how the algorithms perform
- The terms n^2 and n^3 are even simpler and preferred
- These terms can be obtained through **asymptotic analysis**

Asymptotic Analysis

- An analysis of algorithms that focuses on
  a. Analyzing problems of **large input size**
  b. Consider **only the leading term** of the formula
  c. **Ignore the coefficient** of the leading term

Why Choose Leading Term?

- Lower order terms contribute lesser to the overall cost as the input grows larger
- Example
  - $f(n) = 2n^2 + 100n$
  - $f(1000) = 2(1000)^2 + 100(1000)$
    $= 2{,}000{,}000 + 100{,}000$
  - $f(100000) = 2(100000)^2 + 100(100000)$
    $= 20{,}000{,}000{,}000 + 10{,}000{,}000$
- Hence, lower order terms can be ignored

Examples: Leading Terms

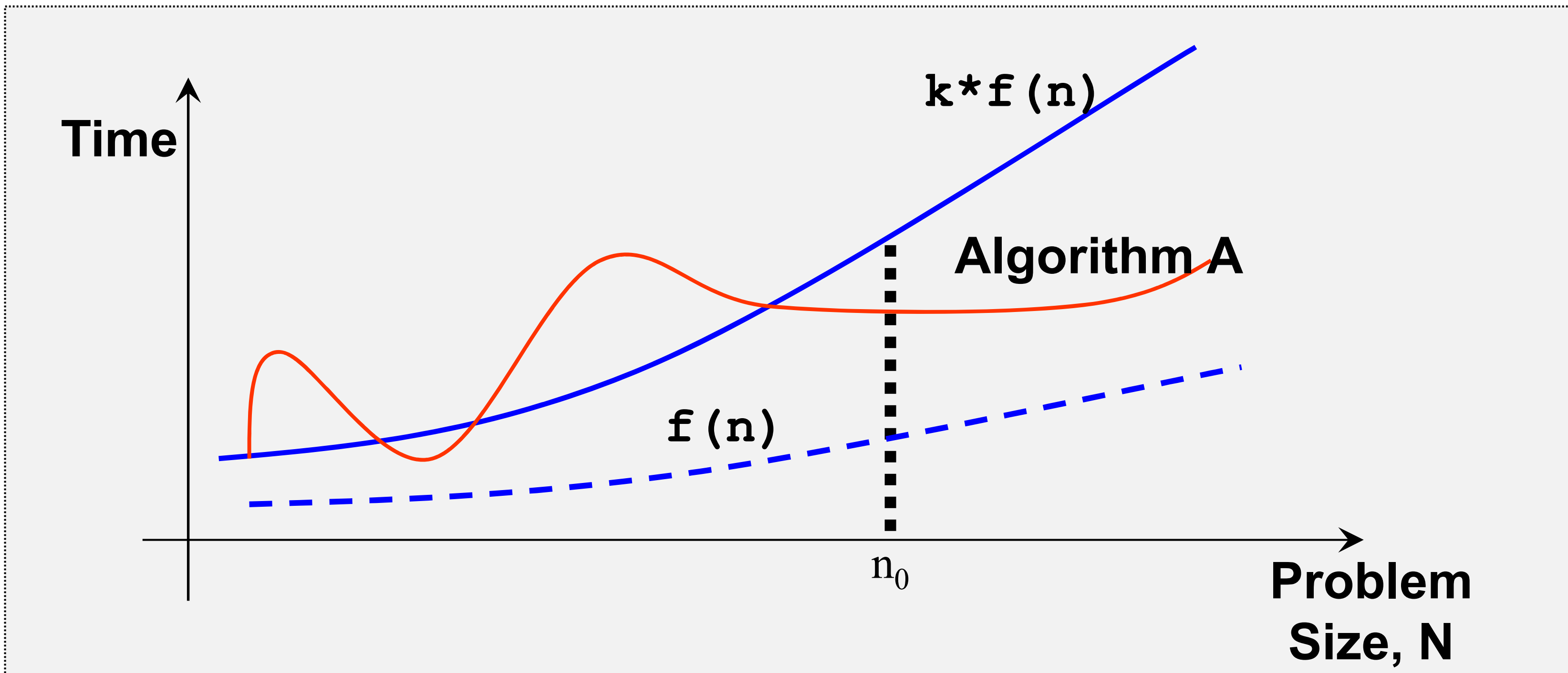
- $a(n) = \frac{1}{2} n + 4$
  - Leading term: $\frac{1}{2} n$
- $b(n) = 240n + 0.001n^2$
  - Leading term: $0.001n^2$
- $c(n) = n \lg(n) + \lg(n) + n \lg(\lg(n))$
  - Leading term: $n \lg(n)$
  - Note that $\lg(n) = \log_2(n)$

Why Ignore Coefficient of Leading Term?

- Suppose two algorithms have $2n^2$ and $30n^2$ as the leading terms, respectively
- Although actual time will be different due to the different constants, the **growth rates** of the running time are the same
- Compare with another algorithm with leading term of n^3, the difference in growth rate is a much more dominating factor
- Hence, we can drop the coefficient of leading term when studying algorithm complexity

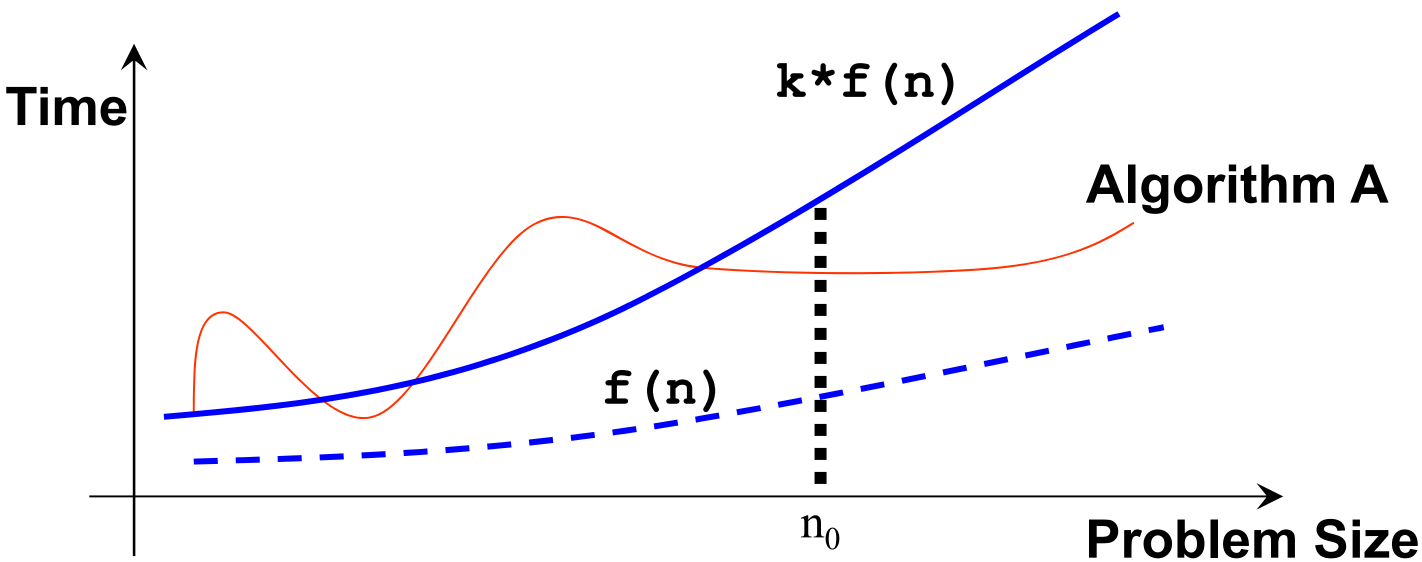
The Big-O Notation: Definition

Algorithm *A* is of **O(*f*(*n*))**
if there exist a **constant *k***, and a **positive integer n_0**
such that Algorithm *A* requires
no more than *k* * *f*(*n*) time units to
solve a **problem of size *n* >= n_0**



The Big-O Notation

- When problem size is larger than n_0, Algorithm *A* is **bounded from above** by $k * f(n)$
- Observations
  - n_0 and k are **not unique**
  - There are many possible $f(n)$

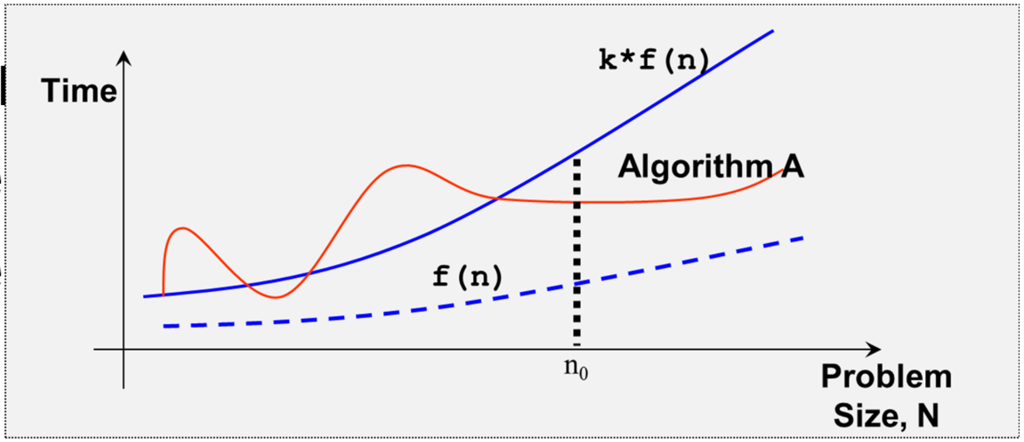


Example: Finding n_0 and k

- Given complexity of Algorithm *A* is $2n^2 + 100n$
- **Claim:** Algorithm *A* is of $O(n^2)$

- **Solution:**
  - $2n^2 + 100n < 2n^2 + n^2 = \mathbf{3n^2}$ whenever $\mathbf{n > 100}$
  - Set the constants to be $\mathbf{k = 3}$ and $\mathbf{n_0 = 101}$
  - By definition, we say **Algorithm A** is $\mathbf{O(n^2)}$

- **Questio**
  - Can we
  - Can we



Growth Terms

- By asymptotic analysis, it is clear that:
  - Coefficient of the **f(n)** can be absorbed into the constant k
  - E.g. **A** is `O(3n²)` with constant k_1

  ➔ **A** is `O(n²)` with constant $k = k_1 * 3$
  - So, **f(n)** can be reduced to function with **coefficient of 1** only
- Such a term is called a **growth term**
- Ordered list of the commonly seen **growth terms**:

$O(1) < O(\lg(n)) < O(n) < O(n \lg(n)) < O(n^2) < O(n^3) < O(2^n)$

"fastest" ***"slowest"***

- "lg" = $\log_2$
- In big-O, log functions of different bases are all the same (why?)

Common Growth Rates

- **O(1) — constant time**
  - Independent of *n*
- **O(*n*) — linear time**
  - Grows as the same rate of *n*
  - E.g. double input size ➔ double execution time
- **O(n^2) — quadratic time**
  - Increases rapidly w.r.t. *n*
  - E.g. double input size ➔ quadruple execution time
- **O(n^3) — cubic time**
  - Increases even more rapidly w.r.t. *n*
  - E.g. double input size ➔ 8 * execution time
- **O(2^n) — exponential time**
  - Increases very very rapidly w.r.t. *n*

Example: Exponential-Time Algorithm

- Algorithm A:
  - For an input of **n** items
  - Can be solved by going through 2^n cases

- We use a supercomputer*, that analyses *200* million cases per second
  - Input with *15* items, ***163*** **microseconds**
  - Input with *30* items, ***5.36*** **seconds**
  - Input with *50* items, **more than two months**
  - Input with *80* items, ***191*** **million years**

Example: Quadratic-Time Algorithm

- Algorithm B:
  - input of **n** items need to go through **$300n^2$** *cases*

- *Handheld PC*, running at *33* MHz
  - Input with *15* items, ***2*** **milliseconds**
  - Input with *30* items, ***8*** **milliseconds**
  - Input with *50* items, ***22*** **milliseconds**
  - Input with *80* items, ***58*** **milliseconds**

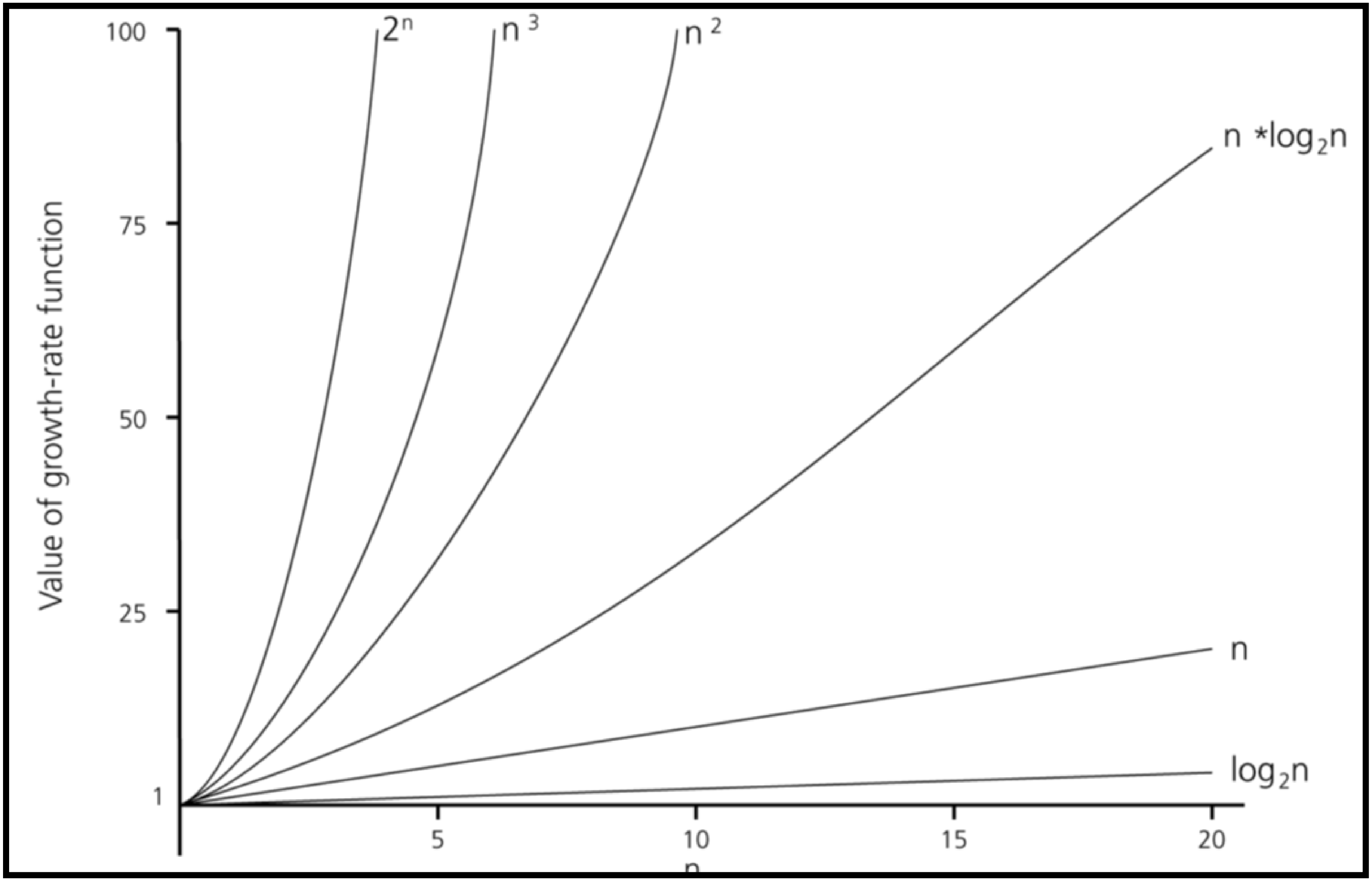
- Don't depend on the raw power of a computer to speed up program!

- It is very important to use an **efficient algorithm** to solve a problem

Comparing Growth Rates

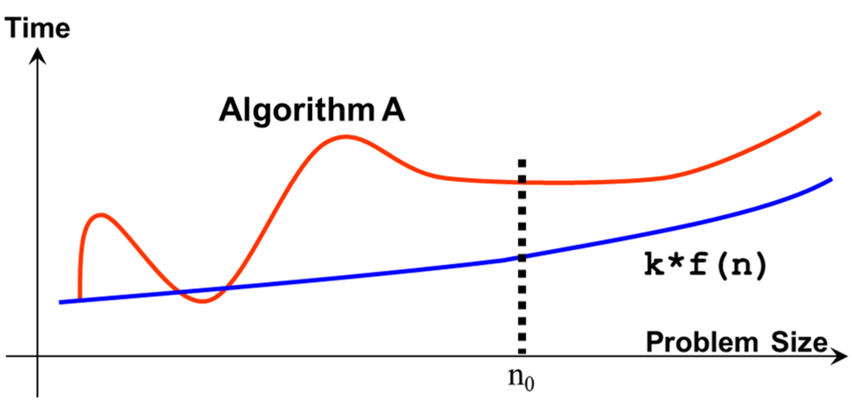
Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Comparing Growth Rates

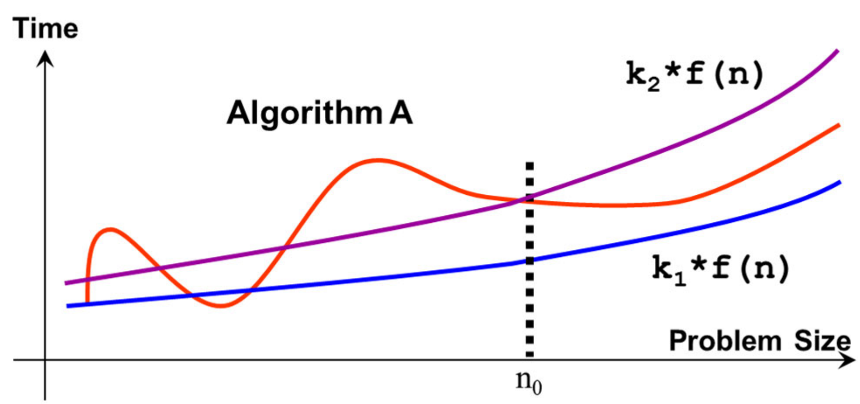


Big-Omega & Big-Theta

- Similar to Big-O notation:
  - Big-Omega: Establishes the **lowerbound** of the growth rate of the algorithm
  - Big-Theta: Establishes the **tightbound** of the growth rate of the algorithm
- We focus mainly on Big-O in this course



$\Omega(f(n))$



$\Theta(f(n))$

Example: Finding Complexity (1/2)

- What is the complexity of the following code fragment?

```
i = 1
while i <= n:
    i = i*2
    #other simple operations
```

- The loop executes:

  i = 1, 2, 4, 8, …, 2^k where $k = \lfloor \log_2 \boldsymbol{n} \rfloor$

  There are k + 1 iterations

  So the complexity is O(k), i.e. O(log n)

Example: Finding Complexity (2/2)

- What is the complexity of the following code fragment?

```
for i in range(1, n+1):                #outer loop
    for j in range(1, n+1):            #inner loop
        print(str(i*j) + ' ', end='')
    print("\n")
```

- Each value of "**i**": the inner loop runs **n** times
- In total: n values of **i**: **n * n = n²**

➔ Complexity is $O(n^2)$

Analysis 1: Sequential Search

- Check whether an item **target** is in an unsorted array
  a. If found, it returns **position (index)** of **target** in array
  b. If not found, it returns **-1**

```
def seqSearch(array, target):
    for i in range(len(array)):
        if array[i] == target:
            return i
    return -1
```

Analysis 1: Sequential Search

- Time spent in each iteration through the loop is at most some constant c_1
- Time spent outside the loop is at most some constant c_2
- **Maximum** number of iterations is n
- Hence, the asymptotic upper bound is $c_1 n + c_2 = O(n)$
- Observation
  - In general, a loop of n iterations will lead to $O(n)$ growth rate
  - This is an example of **Worst Case Analysis**

Analysis 2: Binary Search

- Important characteristics
  - Requires array to be sorted
  - Maintain sub-array where *target* might be located
  - Repeatedly compare *target* with *m*, the middle of current sub-array
    - If *target* = *m*, found it!
    - If *target* > *m*, eliminate *m* and positions before *m*
    - If *target* < *m*, eliminate *m* and positions after *m*

- Iterative and recursive implementations

Binary Search (Iterative)

```python
def binSearch(array, target):
    left = 0
    right = len(array)-1
    while left <= right:
        middle = (left + right) // 2
        if array[middle] < target:
            left = middle + 1
        elif array[middle] > target:
            right = middle - 1
        else:
            return middle
    return -1
```

Binary Search (Recursive)

```python
def binSearch(array, target):
    if array == []:
        return -1

    middle = len(array) // 2
    if array[middle] < target:
        return binSearch(array[:middle], target)
    elif array[middle] > target:
        return binSearch(array[middle+1:], target)
    else: #found it!
        return middle
```

Analysis 2: Binary Search (Iterative)

- Time spent outside the loop is at most c_1
- Time spent in each iteration of the loop is at most c_2
- For inputs of size n, if the program goes through at most $f(n)$ iterations, then the complexity is

  $c_1 + c_2 f(n)$ or $O(f(n))$

- i.e. the complexity is decided by the number of iterations (loops)

Analysis 2: Finding $f(n)$

- At any point during binary search, part of array is "alive" (might contain x)
- Each iteration of loop eliminates at least half of previously "alive" elements
- At the beginning, all n elements are "alive", and after
  - One iteration, at most $n/2$ are left, or alive
  - Two iterations, at most $(n/2)/2 = n/4 = n/2^2$ are left
  - Three iterations, at most $(n/4)/2 = n/8 = n/2^3$ are left
  - . . .
  - k iterations, at most $n/2^k$ are left
  - At the final iteration, at most 1 element is left

Analysis 2: Finding $f(n)$

- In the **worst case**, we have to search all the way up to the last iteration k with only one element left
- We have

$$n/2^k = 1 \quad \Rightarrow \quad 2^k = n \quad \Rightarrow \quad k = \log_2(n) = \lg(n)$$

- Hence, the binary search algorithm takes $O(f(n))$, or $O(\lg(n))$ time
- Observation
  - In general, when the domain of interest is reduced by a fraction for each iteration of a loop, then it will lead to $O(\log n)$ growth rate

Analysis of Different Cases

- For an algorithm, three different cases of analysis
  - **Worst-Case Analysis**
    - Look at the worst possible scenario
  - **Best-Case Analysis**
    - Look at the ideal case
    - Usually not useful
  - **Average-Case Analysis**
    - Probability distribution should be known
    - Hardest/impossible to analyze
- Example: Sequential Search
  - **Worst-Case**: target item at the tail of array
  - **Best-Case**: target item at the head of array
  - **Average-Case**: target item can be anywhere

Space Complexity?

- The idea of "Time Complexity" can be applied to **"Space Complexity"**:
  - Focus on the amount of memory usage w.r.t. n

- Memory usage comes from:
  - Variables
  - Data structures like array, list etc
  - Function calls*
    - Each function call create new set of local variables in most languages

Example: Space Complexity (1/2)

- Relook at this example and focus on **space complexity**

```
for i in range(1, n+1):                 #outer loop
    for j in range(1, n+1):             #inner loop
        print(str(i*j) + ' ', end='')
    print("\n")
```

- Only "**i**" and "**j**" variable are created / used, regardless of **n**

→ **O(1) space complexity**

- What it means:
  - Regardless of the **n** value, this program will use the **same amount of memory**

Example: Space Complexity (2/2)

- Relook at this example and focus on **space complexity**

```
matrix = []
for i in range(0, n+1):
    matrix.append([])  #add a new empty row
    for j in range (0, n+1):
        matrix[i].append(i*j)
```

- In the end, there are n rows, each with n values, i.e. n^2 values stored in matrix

➔ Space complexity of **O(n^2)**

- What it means:
  - With larger **n** value, this program will use the much more memory!

Summary

- Algorithm Definition
- Algorithm Analysis
  - Counting operations
  - Asymptotic Analysis
  - Big-O notation (Upper-Bound)
- Three cases of analysis
  - Best-case
  - Worst-case
  - Average-case
- Space Complexity

END