UVA DATA SCIENCE

Foundation of Computer Science for Data Science

# Computational Complexity

Mai Dahshan

September 2, 2024

# Learning Objectives

- Introduction to data structures and algorithms
- Understand how to measure performance of algorithms
- Understand the asymptotic analysis of algorithms
- Understand the difference between orders of growth

# Data Structure

"Get your data structures correct first, and the rest of the program will write itself."

- *David Jones*

# Data Structure

- A *Data Structure* is:
  - an implementation of an ADT *and*
  - is a way of organizing and storing data in a computer so that it can be accessed and manipulated efficiently.

# Data Structure

- Data Structures will have 3 core operations
    - a way to add things
    - a way to remove things
    - a way to access things
- Details of these operations depend on the data structure
    - Example: List, add at the end, access by location, remove by location
- More operations added depending on what data structure is designed to do

# Data Structure

- why so many  data structures?
- The study of data structures is the study of tradeoffs.  That's why we have so many of them!
  - time *vs.* space
  - generality *vs.* simplicity

# Algorithm

- An **algorithm** is a detailed step-by-step method for solving a problem
- *Properties of algorithms*
  - Steps are precisely stated
  - Determinism: based on inputs, previous steps
  - Algorithm terminates
  - Also: correctness, generality, efficiency, usability, robustness, debuggability

# Algorithm

- How to describe an algorithm?
  - Problem Definition: objective, inputs, outputs
  - Algorithm Steps: initialization, processing, termination
  - Pseudo code
  - Flow chart

# Algorithm

## Algorithm

**Purpose:** To find the smallest value in a list of numerical values.

1. **Check if the List is Empty:** If the list is empty, return an error message
2. **Initialize the Minimum Value:** Set the initial minimum value to the first element of the list
3. **Iterate Through the List:** Traverse each element of the list starting from the second element
4. **Compare and Update Minimum Value:**
    i. For each element, compare it with the current minimum value
    ii. If the element is smaller than the current minimum value, update the minimum value
5. **Return the Minimum Value:** After traversing all elements, return the minimum value

## Pseudo code

```
Input: list - a list of numerical values
Output: minValue - the smallest value in the list

if length(list) == 0 then
    return "Error: The list is empty"   // or some other indication of an
empty list
end if

minValue = list[0]  // Assume the first element is the minimum

for i from 1 to length(list) - 1 do
    if list[i] < minValue then
        minValue = list[i]
    end if
end for

return minValue
```

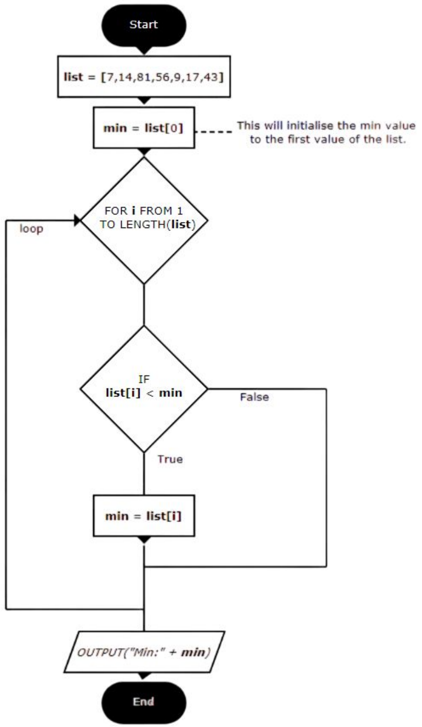
# Algorithm

## Algorithm

**Purpose:** To find the smallest value in a list of numerical values.

1. **Check if the List is Empty:** If the list is empty, return an error message
2. **Initialize the Minimum Value:** Set the initial minimum value to the first element of the list
3. **Iterate Through the List:** Traverse each element of the list starting from the second element
4. **Compare and Update Minimum Value:**
   i. For each element, compare it with the current minimum value
   ii. If the element is smaller than the current minimum value, update the minimum value
5. **Return the Minimum Value:** After traversing all elements, return the minimum value

## Flowchart



# Algorithm Efficiency

how much time or space you use! is most important

- The efficiency of an algorithm is a **measure of the amount of resources consumed in solving a problem of size n**.
    - CPU (time) usage, memory usage, disk usage,network usage,....
- In general, what we care about is
    - Time Efficiency (How much time our algorithm takes to solve problem of size n?)
    - Memory Efficiency (How much memory does our algorithm use to solve problem of size n?)

# Algorithm Efficiency

- We need to estimate the time and memory requirements of the algorithm.
  - Memory is cheap and abundant, making space complexity analysis less common
  - Time is “expensive”. This is why our **primary focus will be on analyzing time complexity**.

*“The biggest difference between time and space is that you can't reuse time.”*

*- Merrick Furst*

# Ways to Measure Algorithm Efficiency

- Empirical Analysis (i.e, Benchmarking) requires implementation
- Asymptotic Analysis mathematical way to eval efficiency as n goes to ininity, not require implementation

# Ways to Measure Algorithm Efficiency

- **Benchmarking** involves **implementing the algorithm** and then assessing its efficiency by **executing it with specific inputs** and **measuring the amount of processor time** required to produce the correct result.

# Ways to Measure Algorithm Efficiency

- Benchmarking has several limitations:
  - to benchmark an algorithm, it is necessary to implement and test it.
  - it is performed using the same hardware and software environments.
  - it measures efficiency for a specific case and may not predict performance with different datasets.
  - it is not suitable for mathematically analyzing the general properties of algorithms.

# Ways to Measure Algorithm Efficiency

- Our goal is to develop an approach to analyzing algorithm efficiency that:
  - takes into account all possible inputs
  - dependent on the nature of the input (best-case, worst-case, and average)
  - evaluates the efficiency of an algorithm independent of the hardware and software environment
  - is performed by studying a high-level description of the algorithm without needing implementation

# Ways to Measure Algorithm Efficiency

- Asymptotic analysis is a method used to describe the **efficiency of algorithms as the input size grows towards infinity**.

# Asymptotic Analysis

- For an algorithm with input size n , define running time T(n) as
  - the amount of time an algorithm takes to complete (time complexity) as a function of the input size n
- The purpose of asymptotic analysis is to evaluate the rate of growth of T(n) as n approaches infinity. mathematically look at rate of growth of n as it goes to efficiency

# Asymptotic Analysis

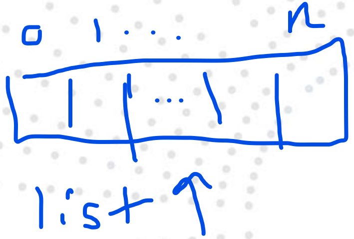
- An pseudocode of an algorithm for searching an element in a list of size n

```
SequentialSearch(ArrayA, targetElement)
   for i = 1 to ArraySize(ArrayA) do
       if ArrayA[i] == targetElement
           return i
   return -1
```

**What is the running of this algorithm?**

# Asymptotic Analysis

- An pseudocode of an algorithm for searching an element in a list of size n




```
SequentialSearch(ArrayA, targetElement)
   for i = 1 to ArraySize(ArrayA) do
       if ArrayA[i] == targetElement
           return i
   return -1
```

**What is the running time of this algorithm?**

**when? Best case? Average case? Or worst case?**

# Different Scenarios of Algorithm Performance

- **Best case**: Minimum operations (i.e., executable statements) required; fastest execution time. like looking for 100 in the list and it is the first element, as it checks element by element
- **Average case**: Expected performance over all possible inputs; typical scenario. 100 is the middle element, and it checks each element
- **Worst case**: Maximum operations required; slowest execution time.

  like looking for 100 in a list and it is the last element in the list

# Asymptotic Analysis

- An pseudocode of an algorithm for searching an element in a list of size n

**What is the running time of this algorithm in best case?** reframed the question

```
SequentialSearch(ArrayA, targetElement)
    for i = 1 to ArraySize(ArrayA) do
        if ArrayA[i] == targetElement
            return i
    return -1
```

| Cost | Times |
|---|---|
| c1 | 1 |
| c2 | 1 |
| c3 | 1 |

T(n) = c1 + c2 + c3

The running time is constant independent of size of list (n)

# Asymptotic Analysis

- An pseudocode of an algorithm for searching an element in a list

**What is the running time of this algorithm in worst case?**

element in the last spot or it is not in the list, iterate through whole array

```
SequentialSearch(ArrayA, targetElement)
    for i = 1 to ArraySize(ArrayA) do
        if ArrayA[i] == targetElement
            return i
    return -1
```

| Cost | Times |
|---|---|
| c1 | $\sum_{i=1}^{n} 1 = n$ |
| c2 | $\sum_{i=1}^{n} 1 = n$ |
| c3 | 1 |
| c4 | 1 |

go whole list

only implemented when found

in list | not in list

T(n) = c1 . n + c2 . n + c3 or T(n) = c1 . n + c2 . n + c4

The running time is linear function of n

# Asymptotic Analysis

- An pseudocode of an algorithm to calculate the sum of all elements in a list of size n

```
SumListElements(listA)
total = 0
for i = 1 to length(list)
    total += list[i]
print(total)
```

**What is the running time of this algorithm in best, average, and worst case?** always frame like this

iterate over whole array in every case because it is a sum!

# Asymptotic Analysis

- Given an algorithm to algorithm to calculate the sum of all elements in list of size n. **what is the running time of this algorithm in best/average/worst cases?**

THE SAME IN EVERY CASE

| SumListElements(listA) | Cost | Times | |
|---|---|---|---|
| total = 0 | c1 | 1 | runs once |
| for i = 1 to length(list) | c2 | $\sum_{i=1}^{n} 1 = n$ | over whole array |
|     total += list[i] | c3 | $\sum_{i=1}^{n} 1 = n$ | over whole list |
| print(total) | c4 | 1 | prints once |

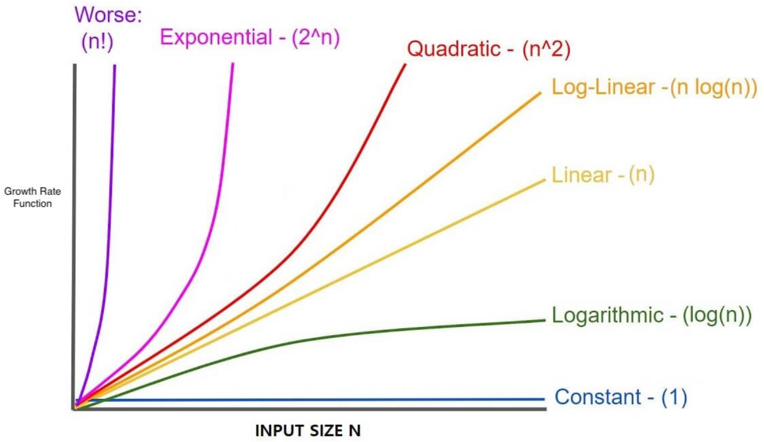
$$T(n) = c1 + c2\,n + c3\,n + c4$$

$$= c1 + c4 + (c2 + c3)n$$

The running time is a linear function of n

# Common Growth Rates

- The asymptotic growth rate for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.




**Rule of thumb:** the slower the asymptotic growth rate, the better the algorithm

very slow

| Function | Common Name |
|---|---|
| $n!$ | Factorial |
| $2^n$ | Exponential |
| $n^d, d > 3$ | Polynomial |
| $n^3$ | Cubic |
| $n^2$ | Quadratic |
| $n\sqrt{n}$ | n Square root n |
| $n \log n$ | n log n |
| $n$ | Linear |
| $\sqrt{n}$ | Root - n |
| $\log n$ | Logarithmic |
| $1$ | Constant |

Fast

# Common Growth Rates - Discussion

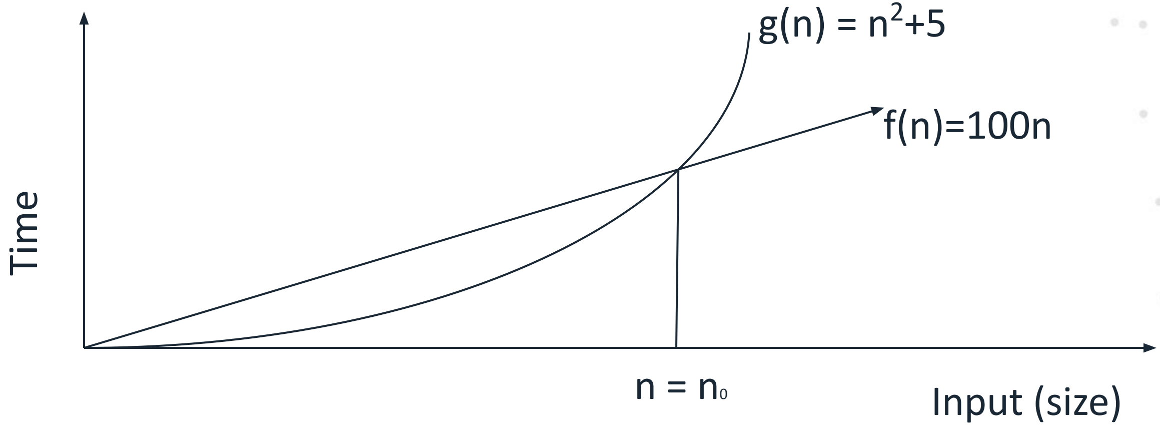
- Given two functions

$$f(n) = 100n \qquad\qquad g(n) = n^2 + 5$$

- When n is small, which is the bigger function? n=5, g is better, smaller
- When n is large, which is the bigger function? n=100, f is slightly better, smaller
- can we say *g(n) grows faster than f(n)?* NOOOO

# Common Growth Rates - Discussion

- As size of input (n) gets larger, any algorithm of a smaller order will be more efficient than an algorithm of a larger order

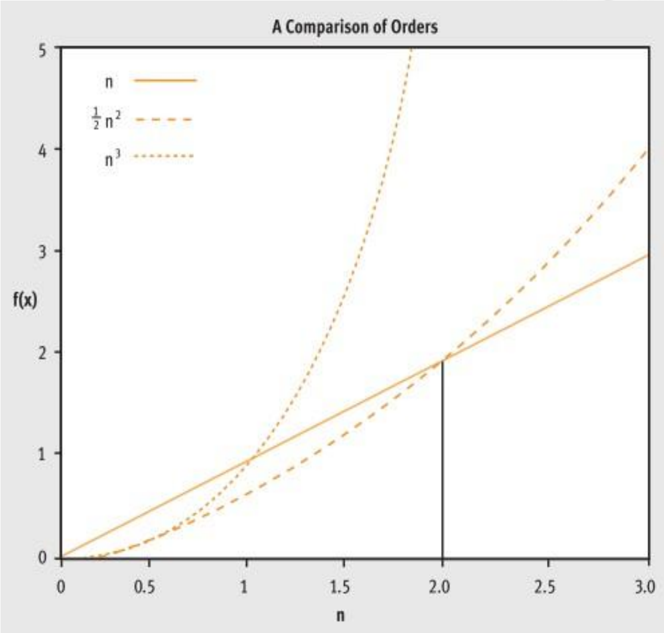


# Asymptotically Superior Algorithm

- If we choose an **asymptotically superior algorithm** to solve a problem, we will not know exactly how much time is required, but we know that as the problem size increases there will always be a point beyond which the lower-order method takes less time than the higher-order algorithm
- Once the problem size becomes sufficiently large, the asymptotically superior algorithm always executes more quickly
- The next figure demonstrates this behavior for algorithms of order $n$, $n^2$, and $n^3$

# Asymptotically Superior Algorithm

- For small problems, the choice of algorithms is not critical – in fact, the growth rate $n^2$or $n^3$ may even be superior!
- However, as n grows large (larger than **2.0** in this case) the algorithm with growth rate of **n** always has a superior running time and ***improves as n increases***



# Asymptotic Notations

trying to measure efficiency as n goes to infinity

Trying to bound the efficiency

- **Asymptotic notations** are mathematical tools used to express the growth rate, allowing us to classify and compare algorithms based on their efficiency
- They help us to reason about our algorithms with various cases like best case, worst case, and average case.
- Three main asymptotic notations
  - Big-O Notation (O-notation) - Upper bound
  - Omega Notation (Ω-notation) - Lower bound
  - Theta Notation (Θ-notation) - Tight bound

bounds between upper (best efficiency and a lower bound (worst)

# Big-O Notation

t could = f but not have to

- Given two functions t(n) & f(n) for input n, we say t(n) is in O(f(n) ) iff there exist positive constants c and $n_0$ such that

is there a function bounding it, so my function is never bigger

$$t(n) \leq c\, f(n) \text{ when } n \geq n_0$$

n0 is a value for n

  - n is the size of the dataset the algorithm works on
  - t(n) is the actual growth rate of the algorithm
  - f(n) is a function that characterizes an upper bounds on t(n)
    - it is the function that bounds the growth rate (It is a limit on the running time of the algorithm)
  - c and $n_0$ are constants. The choices for c and $n_0$ are not unique

# Big-O Notation

fn and

need this to hold:
T(n)<=cf(n) for this case make them the same:
5n+2<=c(5n+2) where c is 1 and n is 1
or like 100n<= c(n^2+5)

- Example
  - $t(n) = 5n+2 = O(f(n)) = O(n)$ for what value of c and $n_0$?
    - $t(n) \leq 6n$, for $n \geq 3$ $(c=6, n_0=3)$
  - $t(n) = n/2 - 3 = O(f(n)) = O(n)$ for what value of c and $n_0$?
    - $t(n) \leq 0.5\, n$ for $n \geq 0$ $(c=0.5, n_0=0)$
  - $t(n) = n(n+1)/2 = O(f(n)) = O(n^2)$ for what value of c and $n_0$?
    - $n(n+1)/2 \leq n^2$ for $n \geq 0$ $(c=1, n_0=0)$

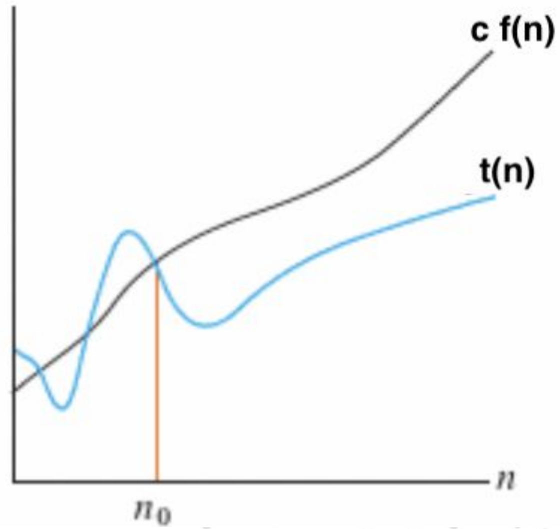
looking for a starting point n0 so that as n goes to infinity, the equation is satisfied

tight bound is better than loose bound, as long as satisfying the equation you are good

minimum upper bound is O(n)

# Big-O Notation

- t(n) is O(f(n)) if t(n) is asymptotically **less than or equal** to f(n)
    - it is correct to say 7n - 3 is $O(n^2)$ or $O(n^3)$, a better statement is 7n - 3 is O(n), make the approximation as tight as possible
- t(N) may not necessarily equal f(N)
    - constants and lesser terms ignored because it is a *bounding function*




t(n) = O(f(n))

if function describing growth rate, take the highest term

note starting at n0, then cf(n) bounds t(n)

# Big-O Notation

- Big-Oh Rules
  - If a function that describes the growth of an algorithm has several terms, its order of growth is determined by the **fastest growing term**
    - If $\boldsymbol{t}(\boldsymbol{n})$ is a polynomial of degree $\boldsymbol{d}$, then $\boldsymbol{t}(\boldsymbol{n})$ is $\boldsymbol{O}(\boldsymbol{n}^{\boldsymbol{d}})$, (i.e., $a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0 = O(n^m)$)
      - Drop lower-order terms
      - Drop constant factors
      - Example: $7n^3 + 20\,n^2 + 4\,n$ is $O(n^3)$
  - Use the smallest possible class of functions
    - Say "$2\boldsymbol{n}$ is $\boldsymbol{O}(\boldsymbol{n})$" instead of "$2\boldsymbol{n}$ is $\boldsymbol{O}(\boldsymbol{n}^2)$"
  - Use the simplest expression of the class
    - Say "$3\boldsymbol{n} + 5$ is $\boldsymbol{O}(\boldsymbol{n})$" instead of "$3\boldsymbol{n} + 5$ is $\boldsymbol{O}(3\boldsymbol{n})$"

# Big-O Notation

- t(n) = $n^4 + 100n^2 + 10n + 50$
  - t(n) = $O(n^4)$
- t(n) = $10n^3 + 2n^2$
  - t(n) = $O(n^3)$
- t(n) = $n^3 - n^2$
  - t(n) = $O(n^3)$
- t(n) = 10
  - t(n) = $O(1)$

# Big-O Notation

- Caution! Constants sometimes make a difference
  - An algorithm running in time 1,000,000 n is still $\boldsymbol{O}(n)$ but might be less efficient on your dataset than one running in time $2n^2$, which is $\boldsymbol{O}(n^2)$

example find c and a starting n0 so
5n^2+5>=c(2n+10),
let c=2 and solve for n so that this is always true

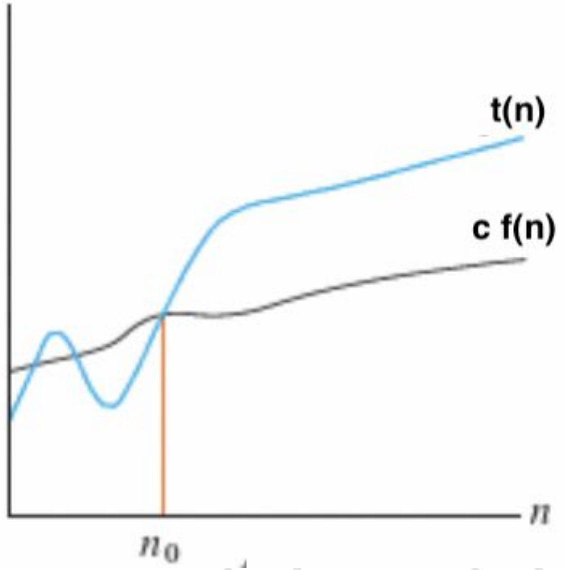
# Big-Omega Notation

omega and big O are inverse

- Given two functions t(n) & f(n) for input n, we say t(n) is in Ω(f(n) ) iff there exist positive constants c and $n_0$ such that

  $t(n) \geq c\, f(n)$ when $n \geq n_0$

  - Big Omega is just opposite to Big O => If t(*n*) is O (f(*n*)) then f(*n*) is Ω(t(*n*))
  - f(n) is a function that characterizes an lower bounds on t(n)
  - t(n) is Ω(f(n)) if t(n) is asymptotically greater than or equal to f(n)




t(n) = Ω(f(n))

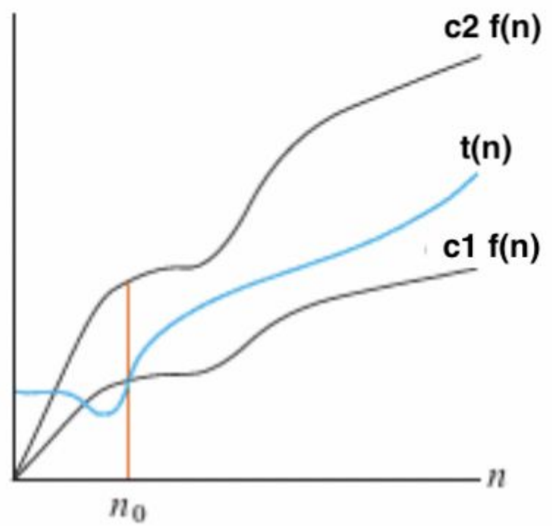
# Big-Theta Notation

tied bound: T(n)=O(n)=omega(n)=theta(n)

- The function t($n$) is $\Theta$(f($n$)) iff there exist two real positive constants $c_1 > 0$ and $c_2 > 0$ and a positive integer $n_0$ such that:

$$c_1 f(n) \geq t(n) \geq c_2 f(n) \text{ for all } n \geq n_0$$

  - For any two functions, f and t, t($n$) = $\Theta$(f($n$)), iff t(n) = O(f($n$)) AND t($n$) =$\Omega$ (f($n$))
  - t(n) is $\Theta$(f(n)) if t(n) is asymptotically **equal** to f(n)




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

# Summary of Asymptotic Notations

| Analysis Type | Mathematical Expression | Relative Rates of Growth |
|---|---|---|
| Big O | T(N) = O( F(N) ) | $T(N) \leq c\, F(N)$ |
| Big Ω | T(N) = Ω( F(N) ) | $T(N) \geq c\, F(N)$ |
| Big θ | T(N) = θ( F(N) ) | $c_1 F(n) \geq T(n) \geq c_2 F(n)$ |

ineff upper bound

tightest

# What does all this means?

- If $t(n) = \Theta(f(n))$ we say that $t(n)$ and $f(n)$ grow at the same rate, asymptotically

same order

- If $t(n) = O(f(n))$ and $t(n) \neq \Omega(f(n))$, then we say that $t(n)$ is asymptotically slower growing than $f(n)$.

like not same order (power)

- If $t(n) = \Omega(f(n))$ and $t(n) \neq O(f(n))$, then we say that $t(n)$ is asymptotically faster growing than $f(n)$.

# Which Notation to Use?

- To express the efficiency of our algorithms which of the three notations should we use?

- As data scientist we generally like to express our algorithms as big O. Why?

because we need to now upper bound because is most inefficient growth rate

# Do Not Be Confused

- Best-Case does not imply $\Omega(f(n))$
- Average-Case does not imply $\Theta(f(n))$
- Worst-Case does not imply $O(f(n))$
- Best-, Average-, and Worst- are specific to the algorithm
- $\Omega(f(n))$, $\Theta(f(n))$, $O(f(n))$ describe functions
    - you can have an $\Omega(f(n))$ bound of the worst-case performance
    - you can have a $\Theta(f(n))$ of best-case

# Common Order Classes - Big O

- O(1) – constant time
- O(lg n) – logarithmic time
- O(n) – linear time
- O(n lg n) – log-linear time
- O($n^2$) – quadratic time
- O($n^3$) – cubic time
- ….
- O($2^n$) – exponential time

Increasing Complexity

# O(1) – Constant time

- The algorithm requires a fixed number of steps regardless of the size of the task (input)
- Examples
  - Push and Pop operations for a stack data structure (size n)
  - Insert and Remove operations for a queue
  - Conditional statement for a loop
  - Assignment statements

# O(1) – Constant time

- listA[Index] O(1) -> Accessing an element in list
- num = 110 O(1) -> Assignment statement
- Conditional Statement

```
if(num>=0): O(1) -> condition check
    print ("num is non negative") O(1) -> print function
else:
    print("num is negative") O(1) -> print function
```

# O(1) – Constant time

```
1   num = 5
2   for i in range(num):
3       print(i)
```

O(1) -> assignment statement

since this is constant, than O(num)=O(1)
if the assignment can be changed or grown,
then it is truly O(num)

O(num) = O(5) = O(1)

O(num) = O(5) = O(1)

If the value of num is fixed (i.e., 5), then time complexity is O(1)

If the value of num can grow to n, then time complexity is O($n$)

# Quiz

What is the worst case time complexity for the following lines? assuming n goes to infinity

```
1  mystr = "a" * n
```

string of n number of a's

interate over the list, so
is O(n)

for i in range(n)
mylist append("a")

```
1  mylist = [0] * n
```

this is also O(n) for the same reason

for i in range(n)
mylist append(0)

# O(log n) – Logarithmic time

- Operations involving dividing the search space in ***half*** each time (taking a list of items, cutting it in half repeatedly until there's only one item left)
- It works for both recursive and non recursive
- Examples
  - Binary search of a sorted list of n elements
  - Insert and Find operations for binary search tree (BST) with n nodes

# O(log n) – Logarithmic time

- This algorithm calculates the number of digits in the binary representation of a number ($n$). The worst case time complexity is O($\log_2 n$)

```
if n <= 0:
    print ("Input must be greater than 0")
count = 0
while n > 0:
    n //= 2
    count += 1
```

O(1) -> condition check

constant time= O(1)

O(1) -> print function

time it takes is independent of the size of my whole program=O(1)

O(1) -> assignment

O($\log_2 n$) -> Each iteration of the loop divides $n$ by 2

each time divides by n/2,n/4,n/8=1 since 3 interations (8=2^3)

$\sum_{i=1}^{\log_2 n} O(1) = \log_2 n \; .O(1) = O(\log_2 n)$

$\sum_{i=1}^{\log_2 n} O(1) = \log_2 n \; .O(1) = O(\log_2 n)$

# O(log n) – Logarithmic time

- An pseudocode of an algorithm calculates the largest power of 2 that is less than or equal to ($n$). The worst case time complexity $O(\log_2 n)$

```
power = 1
while power <= n:



    power *= 2

power = power // 2
```

O(1) -> assignment statement

$O(\log_2 n)$

-> Each iteration of the loop multiplies power by 2 until it exceeds n (2,4,8,16,....)

->number of iterations = num of times power doubled (k)- > starts at 2^0 until it gets 2^k, where k>n

-> 2^k>n = > 2^k≈n => k ≈ $\log_2$n

$\sum_{i=1}^{\log_2 n} O(1) = \log_2 n \,.\, O(1) = O(\log_2 n)$

O(1) -> return statement

# O(log n) – Logarithmic time

- Each recursive call approximately halves the value of n. Thus, the function will continue making recursive calls until n becomes 0.
- The number of recursive calls is approximately **$\log_2 n$**, since each call reduces n by a factor of 2.
- This results in a recursion depth of **$O(\log_2 n)$**

```python
def logn(n):
    if(n==0):
        return ("Done")
    n = n//2
    return logn(n)
```

```python
def largest_power_of_2(n):
    if n <= 1:
        return 1
    half = largest_power_of_2(n // 2)
    return half * 2
```

# O(log n) – Logarithmic time - Quiz

What is the worst case time complexity for the following code snippet?

```
while (n>0)
    n = n / m
```

worst time complexity is: log_m(n)

is n/m^0, n/m^1, n/m^2 etc until n/m^k<=1

# O(n) – Linear time

list, sequential, finding min or max, etc

- The number of steps increase in proportion to the size of the task (input)
- Examples
    - Traversal of a list or an array… (size n)
    - Sequential search in an unsorted list of elements (size n)
    - Finding the max or min element in a list

# O(n) – Linear time

- The function iterates through each element of the array exactly once.
- Therefore, if the size of the array is $n$ , the time complexity of the function for worst case is $O(n)$

Assume arr is of size n

```
def count_occurrences(arr, value):
    count = 0
    for num in arr:
        if num == value:
            count += 1
    return count
```

exec n times

O(1) -> assignment

O(n) -> n iterations in the loop

$\sum_{i=1}^{n} O(1) = n. O(1) = O(n)$

$\sum_{i=1}^{n} O(1) = n. O(1) = O(n)$

O(1) -> return statement

```
def find_max_min(arr):
    if len(arr) == 0:
        print ("List is empty")

    max_val = arr[0]
    min_val = arr[0]

    for element in arr:
        if element > max_val:
            max_val = element
        if element < min_val:
            min_val = element

    return max_val, min_val
```

O(1) -> condition check

O(1) -> print function

O(1) -> assignment

O(1) -> assignment

O(n)

$\sum_{i=1}^{n} O(1) = n. O(1) = O(n)$

$\sum_{i=1}^{n} O(1) = n. O(1) = O(n)$

O(1) -> return statement

add them all up and see the highest order is the overall complexity

# O(n) – Linear time - Quiz

What are the worst case time complexity for the following code snippets?

```
def sequentialSearch(arr, target):
    for element in arr:            O(n)
        return element == target   O(n)
.
```

so overall, most complex is O(n)

```
def sequentialSearch2(arr, target):
    return target in arr
```

O(n)

# O(n lg n) – Log-linear time

- Typically describing the behavior of divide-and-conquer approaches and more advanced sorting algorithms
- **Examples**
  - Quicksort
  - Mergesort

# O(n lg n) – Log-linear time

What are the worst case time complexity for the following pseudocode?

**$O(n \log_2 n)$**

```
y = n
while n > 1:
    n = n // 2
    for i in range(y):
        print(i)
```

$O(1)$ -> assignment statement

$O(\log_2 n)$ -> Each iteration of the loop divides $n$ by 2

$\sum_{i=1}^{\log_2 n} O(1) = \log_2 n \,.\, O(1) = O(\log_2 n)$

$\sum_{i=1}^{\log_2 n} O(n) = \log_2 n \,.\, O(n) = O(n \log_2 n)$

$\sum_{i=1}^{\log_2 n} O(n) = \log_2 n \,.\, O(n) = O(n \log_2 n)$

because for each interaction of outer, inner executes n times

NOTE; complexity of inner loop is complex of outer times inner

# $O(n^2)$ – Quadratic time

- Running time grows proportionally to the square of the input size n
  - For a task of size 10, the number of operations will be 100
  - For a task of size 100, the number of operations will be 100x100 and so on…
- **Examples**
  - A selection sort of n elements
  - Finding duplicates in an unsorted list of size n
  - *Think: doubly nested loops*

# $O(n^2)$ – Quadratic time

What is the worst case time complexity for the following pseudocode?

**$O(n\text{^}2)$**

```
for i in range(n):
    for j in range(n):
        print (i*j)
```

$O(n)$

$\sum_{i=1}^{n} O(n) = n \,.\, O(n) = O(n\text{^}2)$

# $O(n^2)$ – Quadratic time

What is the time complexity of worst case for the following pseudocode?

**O($n$^2)**

each inner iteration going from i to n like 1 to n then 2 to n etc

```
for i in range(n):
    for j in range(i, n):
        print(i*j)
```

O($n$)

$\sum_{i=1}^{n} (n-i) = (n-1) + (n-2) \ldots\ldots + 0$

each going from i to n

= represents the sum of the first $n$ natural numbers

= sum of an arithmetic series

$= \dfrac{n \times (n-1)}{2}$

= O($n$^2)

# $O(n^2)$ – Quadratic time

What is the time complexity of worst case for the following pseudocode?

**$O(n^2)$**

```
for i in range(n):
    for j in range(0, n, 2):
        print(i*j)
```

$O(n)$ from 0 to n-1): 0,1,2,3....n-1

this is 0,2,4,6,,8 etc

$\sum_{i=1}^{n} O(n/2) = n.\ O(n/2) = O\left(\frac{n^2}{2}\right) = O(n^2)$

# O(n²) – Quadratic time

The time complexity for the best-case scenario in insertion sort? **O($n$)**

```
insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >= 0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
```

O($n$)

$\sum_{i=1}^{n-1} O(1) = (n-1).\ O(1) = O(n)$

$\sum_{i=1}^{n-1} O(1) = (n-1).\ O(1) = O(n)$

if the list is already sorted, the body of while loop will not execute at all because arr[j] <= key

$\sum_{i=1}^{n-1} O(1) = (n-1).\ O(1) = O(n)$

Therefore, in the best case, the time complexity is: O($n$)

# $O(n^2)$ – Quadratic time

The time complexity for the worst-case scenario in insertion sort **O(*n*^2)**

```
insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >= 0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
```

$O(n)$

$\sum_{i=1}^{n-1} O(1) = (n-1).\ O(1) = O(n)$

$\sum_{i=1}^{n-1} O(1) = (n-1).\ O(1) = O(n)$

$\sum_{i=1}^{n} o(i-1) = 0 + 1 + 2 \ldots + (n-2) + (n-1)$

= represents the sum of the first $n$ natural numbers =

$= \frac{n \times (n-1)}{2} = O(n^2)$

$\sum_{i=1}^{n-1} O(1) = (n-1).\ O(1) = O(n)$

Therefore, in the average/worst case, the time complexity is: $O(n^2)$

# $O(n^2)$ – Quadratic time - Quizz

The time complexity of insertion sort is O($n$^2) in all cases (best, average, and worst) ?(True or False)

# $O(a^n)$ $(a>1)$ – Exponential time

BAD TO HAVE THIS! like factorial

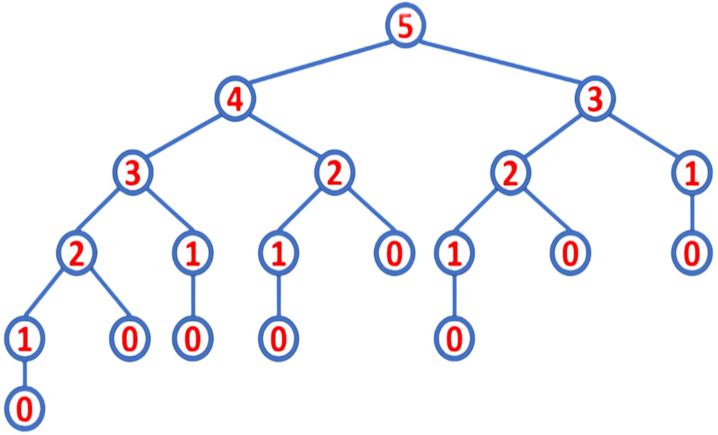
- Many interesting problems fall into this category…
- **Examples**
  - Recursive Fibonacci implementation
  - Towers of Hanoi
  - … many more!

# $O(a^n)$ $(a>1)$ – Exponential time

not dig deeper

- For each call to fibonacci(n), two more calls are made: fibonacci(n - 1) and fibonacci(n - 2). This results in a binary tree of function calls.
- The number of function calls grows exponentiall because the same subproblems are solved multiple times. (the number of calls approximately doubles with each increment in n
- The total number of calls is approximately proportional to 2^$n$ , so the time complexity is $O$ ( 2^$n$)

```python
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```



# Quiz

- For most simple programs or small datasets, efficiency doesn't really matter (True or False) string = immutable

[" Welcome", "to", "UVA" ]

result = " "

1 → welcome

2 → welcome to

3 ⇒ welcome to UVA

# Quiz

WE will start with this!

- For most simple programs or small datasets, efficiency doesn't really matter (True or False)
- But this will not always be true — especially in the data sciences!

```python
#Concatenate a list of strings
#Input: The list of strings to concatenate
#Output: The concatenated string
def concatenate_strings(strings_list):
    return ' '.join(strings_list)
```

checks length, allocates memory, then adds

$O(1)$?

```python
#Concatenate a list of strings
#Input: The list of strings to concatenate
#Output: The concatenated string
def concatenate_strings_with_loop(strings_list):
    result =""                     # O(1)
    for string in strings_list:    # O(n)
        result += string           # n · O(n) = O(n^2)
    return result                  # O(1) · O(n^2) = O(n2)
```

(In memory, adds new space (location) each time)

$\sum^{n} O(n) = n \cdot O(n)$