UVA DATA SCIENCE

Foundation of Computer Science for Data Science

# Computational Complexity

Mai Dahshan

September 2, 2024

UVA DATA SCIENCE

# 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
```

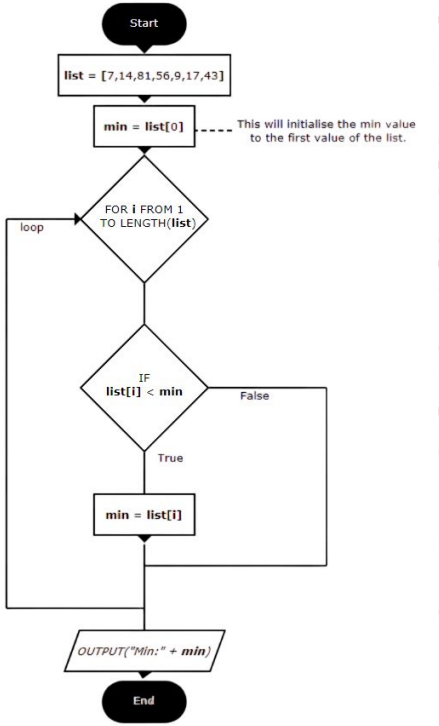
# Algorithm

## 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

- 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)
- Asymptotic Analysis

# 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.

# 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.
- **Average case**: Expected performance over all possible inputs; typical scenario.
- **Worst case**: Maximum operations required; slowest execution time.

# 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?**

```
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?**

```
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 |

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

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)
1   total = 0
2   for i = 1 to length(list)
3         total += list[i]
4   print(total)
```

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

# 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?**

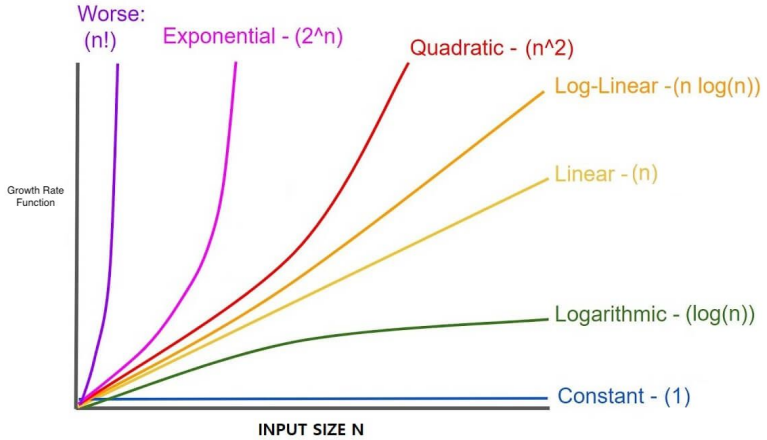
| SumListElements(listA) | Cost | Times |
|---|---|---|
| 1 total = 0 | c1 | 1 |
| 2 for i = 1 to length(list) | c2 | $\sum_{i=1}^{n} 1 = n$ |
| 3 total += list[i] | c3 | $\sum_{i=1}^{n} 1 = n$ |
| 4 print(total) | c4 | 1 |

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

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.




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

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

# Common Growth Rates - Discussion

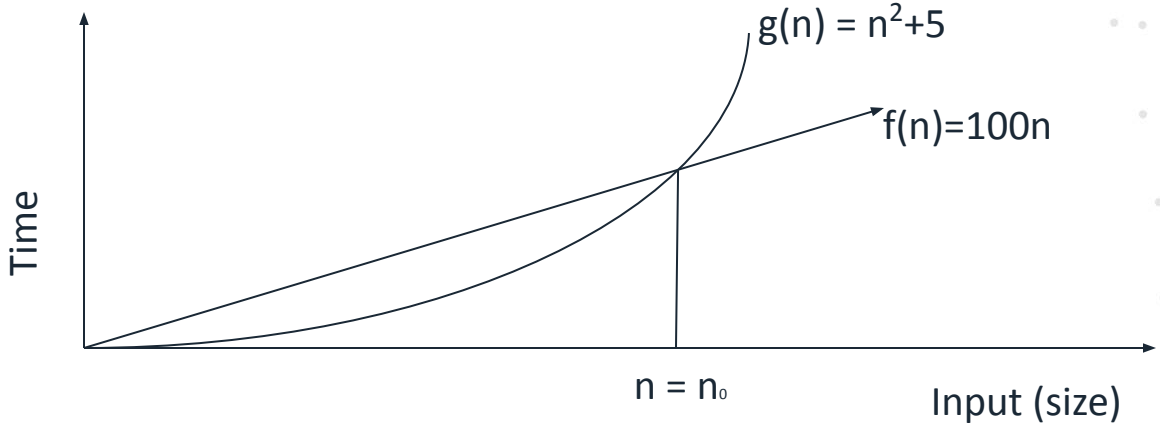
- Given two functions

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

- When n is small, which is the bigger function?
- When n is large, which is the bigger function?
- Can we say *g(n) grows faster than f(n)?*

# 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

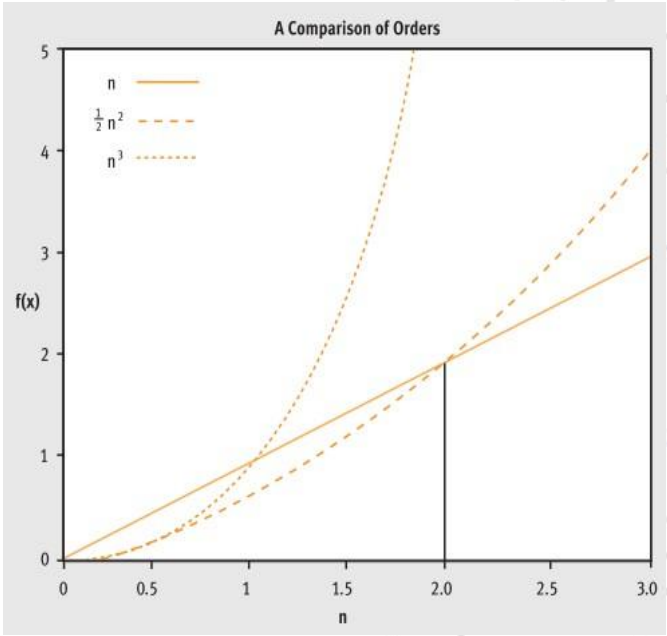


# Asymptotically Superior Algorithm

- 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***



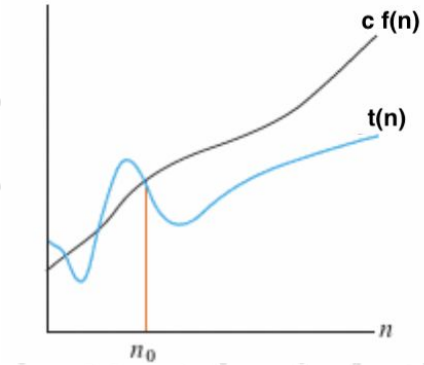
# Asymptotic Notations

- **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

# Big-O Notation

- 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

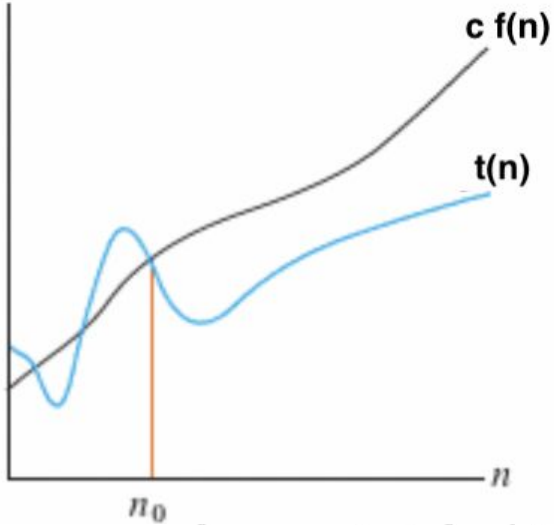
$$t(n) \leq c\, f(n) \text{ when } n \geq n_0$$




  - 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

- 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))

# Big-O Notation

- 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 $n \geq 0$ $(c=1, n_0=0)$

# 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 ***t*(*n*)** is a polynomial of degree ***d***, then ***t*(*n*)** is ***O*(*n*$^{d}$)**, (i.e., $a_m n^m + a_{m-1} n^{m-1} + ... + a_1 n + a_0 = O(n^m)$)
      - 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***n*** is ***O*(*n*)**" instead of "2***n*** is ***O*(*n*$^2$)**"
  - Use the simplest expression of the class
    - Say "3***n*** + 5 is ***O*(*n*)**" instead of "3***n*** + 5 is ***O*(3*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

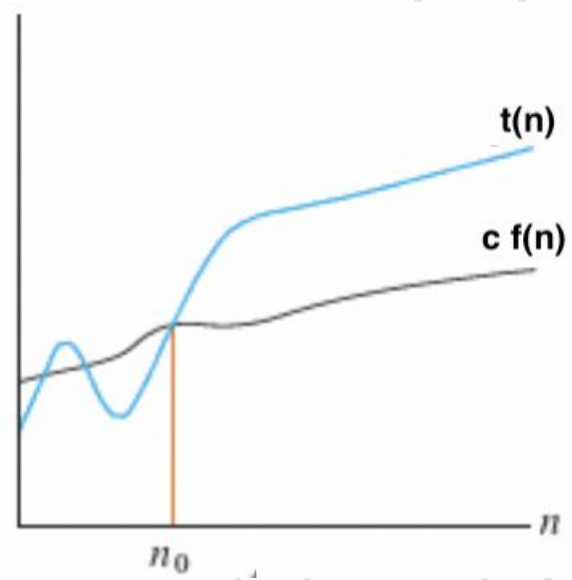
- Caution! Constants sometimes make a difference
  - An algorithm running in time 1,000,000 n is still $\boldsymbol{O}(n)$ but might be less efficient on your dataset than one running in time $2n^2$, which is $\boldsymbol{O}(n^2)$

# Big-Omega Notation

- 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

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

  - 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)



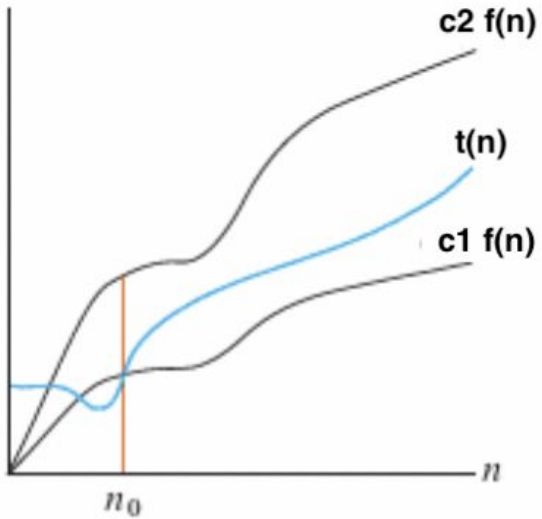

t(n) = Ω(f(n))

# Big-Theta Notation

- The function t($n$) is Θ(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($n$) = Θ(f($n$)), iff t(n) = O(f($n$)) AND t($n$) =Ω (f($n$))
  - t(n) is Θ(f(n)) if t(n) is asymptotically **equal** to f(n)




t(n) = Θ(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 | $T(N) = \Omega(F(N))$ | $T(N) \geq c\,F(N)$ |
| Big θ | $T(N) = \theta(F(N))$ | $c_1 F(n) \geq T(n) \geq c_2 F(n)$ |

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

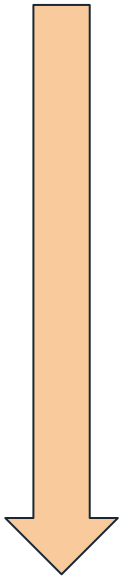
- As data scientist we generally like to express our algorithms as big O. Why?

# Do Not Be Confused

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

```
1 if(num>=0): O(1) -> condition check
2     print ("num is non negative") O(1) -> print function
3 else:
4     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

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?

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

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

# 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 $O(\log_2 n)$

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

O(1) -> condition check

O(1) -> print function

O(1) -> assignment

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

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

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

# 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($\log_2 n$)

```
1  power = 1
2  while power <= n:
3
4
5
6      power *= 2
7
8  power = power // 2
```

power = 1 O(1) -> assignment statement

while power <= n: $O(\log_2 n)$

-> 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 ≈ $\log_2 n$

power *= 2 $\sum_{i=1}^{\log_2 n} O(1) = \log_2 n \cdot O(1) = O(\log_2 n)$

power = power // 2 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 $O(\log_2 n)$

```
1 def logn(n):
2     if(n==0):
3         return ("Done")
4     n = n//2
5     return logn(n)
```

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

# O(log n) – Logarithmic time - Quiz

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

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

# O(n) – Linear time

- 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

```
1  def count_occurrences(arr, value):
2      count = 0
3      for num in arr:
4          if num == value:
5              count += 1
6      return count
```

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

```
1  def find_max_min(arr):
2      if len(arr) == 0:
3          print ("List is empty")
4
5      max_val = arr[0]
6      min_val = arr[0]
7
8      for element in arr:
9          if element > max_val:
10             max_val = element
11         if element < min_val:
12             min_val = element
13
14     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

# O(n) – Linear time - Quiz

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

```
1 def sequentialSearch(arr, target):
2     for element in arr:
3         return element == target
```

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

# 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($n \log_2 n$)**

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

O(1) -> assignment statement

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

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

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

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

# $O(n^2)$ – 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*^2)**

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

O($n$)

$\sum_{i=1}^{n} O(n) = n \cdot O(n) = O(n^2)$

# $O(n^2)$ – Quadratic time

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

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

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

O($n$)

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

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

= sum of an arithmetic series

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

= O($n$^2)

# $O(n^2)$ – Quadratic time

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

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

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

O($n$)

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

# $O(n^2)$ – Quadratic time

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

```
1 insertionSort(arr):
2     for i in range(1, len(arr)):
3         key = arr[i]
4         j = i-1
5         while j >= 0 and key < arr[j]:
6             arr[j+1] = arr[j]
7             j -= 1
8         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)**

```
1 insertionSort(arr):
2     for i in range(1, len(arr)):
3         key = arr[i]
4         j = i-1
5         while j >= 0 and key < arr[j]:
6             arr[j+1] = arr[j]
7             j -= 1
8         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)

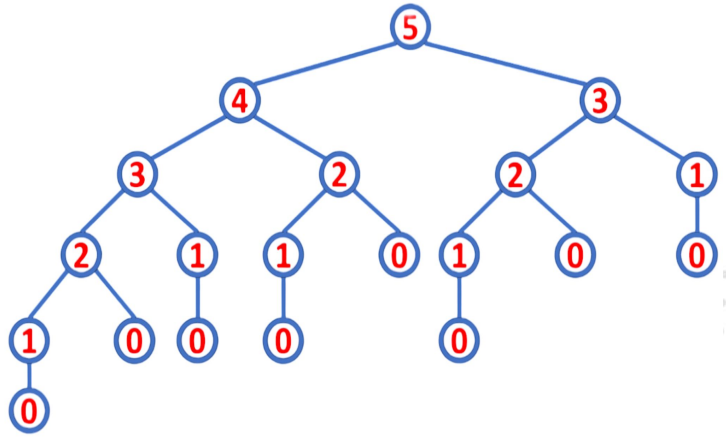
# $O(a^n)$ $(a>1)$ – Exponential time

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

# $O(a^n)$ $(a>1)$ – Exponential time

- 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^$n$)

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



# Quiz

- For most simple programs or small datasets, efficiency doesn’t really matter (True or False)

# Quiz

- 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!

```
#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)
```

```
#Concatenate a list of strings
#Input: The list of strings to concatenate
#Output: The concatenated string
def concatenate_strings_with_loop(strings_list):
    result =""
    for string in strings_list:
        result += string
    return result
```