What is a Data Structure?

A **data structure** (DS) is a way of organizing data so that it can be used effectively.

Why Data Structures?

They are essential ingredients in creating fast and powerful algorithms.

They help to manage and organize data.

They make code cleaner and easier to understand.

Abstract Data Types vs. Data Structures

Abstract Data Type

An **abstract data type** (ADT) is an abstraction of a data structure which provides only the interface to which a data structure must adhere to.

The interface does not give any specific details about how something should be implemented or in what programming language.

Examples

Abstraction (ADT)	Implementation (DS)
List	Dynamic Array Linked List
Queue	Linked List based Queue Array based Queue Stack based Queue
Map	Tree Map Hash Map / Hash Table
Vehicle	Golf Cart Bicycle Smart Car

Computational Complexity Analysis

Complexity Analysis

As programmers, we often find ourselves asking the same two questions over and over again:

How much **time** does this algorithm need to finish?

How much **space** does this algorithm need for its computation?

Big-O Notation

Big-O Notation gives an upper bound of the complexity in the **worst** case, helping to quantify performance as the input size becomes **arbitrarily large**.

Big-O Notation

n - The size of the input
Complexities ordered in from smallest to largest

Constant Time: **O(1)**
Logarithmic Time: **O(log(n))**
Linear Time: **O(n)**
Linearithmic Time: **O(nlog(n))**
Quadric Time: **$O(n^2)$**
Cubic Time: **$O(n^3)$**
Exponential Time: **$O(b^n)$**, $b > 1$
Factorial Time: **O(n!)**

Big-O Properties

$$O(n + c) = O(n)$$

$$O(cn) = O(n),\ c > 0$$

Let f be a function that describes the running time of a particular algorithm for an input of size n:

$$f(n) = 7\log(n)^3 + 15n^2 + 2n^3 + 8$$

$$O(f(n)) = O(n^3)$$

Practical examples coming up don't worry :)

Big-O Examples

The following run in constant time: **O(1)**

```
a := 1
b := 2
c := a + 5*b
```

```
i := 0
While i < 11 Do
    i = i + 1
```

Big-O Examples

The following run in linear time: **O(n)**

```
i := 0
While i < n Do
    i = i + 1
```

$f(n) = n$

$O(f(n)) = O(n)$

```
i := 0
While i < n Do
    i = i + 3
```

$f(n) = n/3$

$O(f(n)) = O(n)$

Big-O Examples

Both of the following run in quadratic time. The first may be obvious since *n* work done *n* times is *n***n* = **O(n²)**, but what about the second one?

```
For (i := 0 ; i < n; i = i + 1)
    For (j := 0 ; j < n; j = j + 1)
```

f(n) = n*n = n², O(f(n)) = **O(n²)**

```
For (i := 0 ; i < n; i = i + 1)
    For (j := i ; j < n; j = j + 1)
                ^ replaced 0 with i
```

Big-O Examples

For a moment just focus on the second loop. Since *i* goes from [0,n) the amount of looping done is directly determined by what *i* is. Remark that if ***i=0***, we do *n* work, if *i=1*, we do *n-1* work, if *i=2*, we do *n-2* work, etc…

So the question then becomes what is:
(n) + (n-1) + (n-2) + (n-3) + … + 3 + 2 + 1?
Remarkably this turns out to be *n(n+1)/2*, so
$O(n(n+1)/2) = O(n^2/2 + n/2) = \mathbf{O(n^2)}$

```
For (i := 0 ; i < n; i = i + 1)
    For (j := i ; j < n; j = j + 1)
```

Big-O Examples

Suppose we have a sorted array and we want to find the index of a particular value in the array, if it exists. What is the time complexity of the following algorithm?

```
low  := 0
high := n-1
While low <= high Do

    mid := (low + high) / 2

    If array[mid] == value: return mid
    Else If array[mid] < value: lo = mid + 1
    Else If array[mid] > value: hi = mid - 1

return -1 // Value not found
```

Ans: $O(\log_2(n)) = \mathbf{O(\log(n))}$

Big-O Examples

Finding all subsets of a set – **O(2^n)**

Finding all permutations of a string – **O(n!)**

Sorting using mergesort – **O(nlog(n))**

Iterating over all the cells in a matrix of size n by m – **O(nm)**

Retrieve an Element from an Array

1. Multiply the size of the element by its index
2. Get the start address of the array
3. Add the start address to the result of the multiplication

Retrieve an Element from an Array

1. Multiply the size of the element by its index
2. Get the start address of the array
3. Add the start address to the result of the multiplication

For an int array, assume element starts at address 12. Each int is 4 bytes.

To get intArray[0] = 12 + 0 * 4 = 12

To get intArray[1] = 12 + 1 * 4 = 16

To get intArray[2] = 12 + 2 * 4 = 20

To get intArray[3] = 12 + 3 * 4 = 24

Number of Elements	Steps to Retrieve
1	3
1000	3
100000	3
1000000	3
1000000000	3

Operation	Time Complexity
Retrieve with index	O(1) – Constant time
Retrieve without index	O(n) – Linear time
Add an element to a full array	O(n)
Add an element to the end of an array (has space)	O(1)
Insert or update an element at a specific index	O(1)
Delete an element by setting it to null	O(1)
Delete an element by shifting elements	O(n)