GAM160: Further Games Programming

6: Data Structures and Collections

Learning outcomes

- **Understand** the various collection classes in C#
- **Compare** the collection classes
- **Implement** an application which uses collection classes

FALMOUTH UNIVERSITY

Common Data Structures

Introduction

Introduction

- In Programming we have concept of reusable data structures which can be used to build applications

Introduction

- In Programming we have concept of reusable data structures which can be used to build applications
- These can be used in order to build larger systems (e.g. Inventory Systems, AI Navigation etc)

Introduction

- In Programming we have concept of reusable data structures which can be used to build applications
- These can be used in order to build larger systems (e.g. Inventory Systems, AI Navigation etc)
- Most programming languages have these built in

Introduction

- In Programming we have concept of reusable data structures which can be used to build applications
- These can be used in order to build larger systems (e.g. Inventory Systems, AI Navigation etc)
- Most programming languages have these built in
- Before writing any system you should always examine these data structures and pick the appropriate one for your Use Case

FALMOUTH UNIVERSITY

Big-O-Notation

What is Big 'O' Notation

What is Big ‘O’ Notation

- The efficiency of an algorithm can be gauged by how long it takes

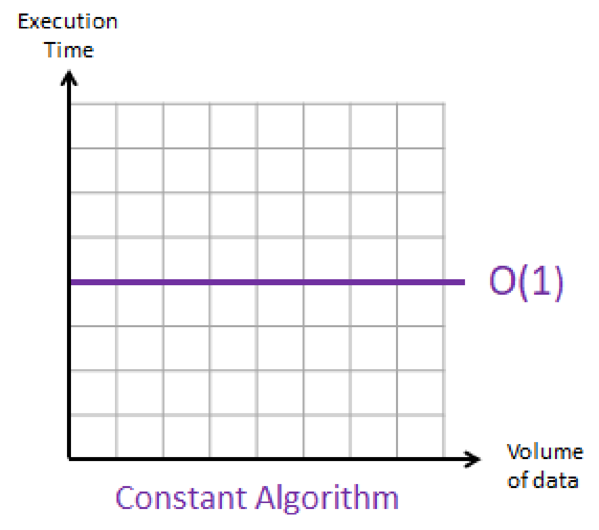
What is Big ‘O’ Notation

- The efficiency of an algorithm can be gauged by how long it takes
- This is know as **Time Complexity**

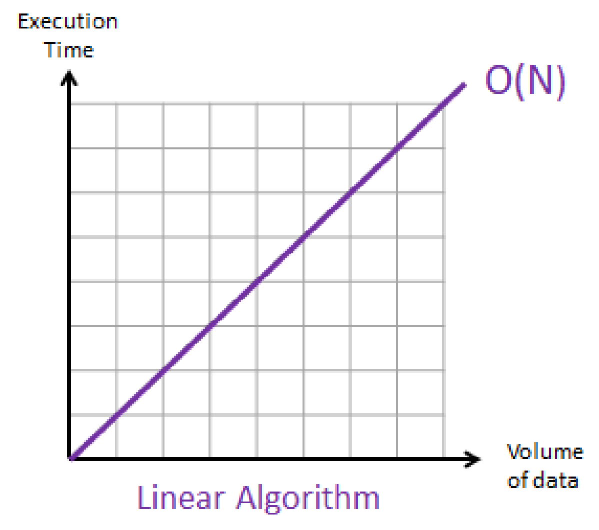
What is Big 'O' Notation

- The efficiency of an algorithm can be gauged by how long it takes
- This is know as **Time Complexity**
- **Big O Notation** is used to describe this

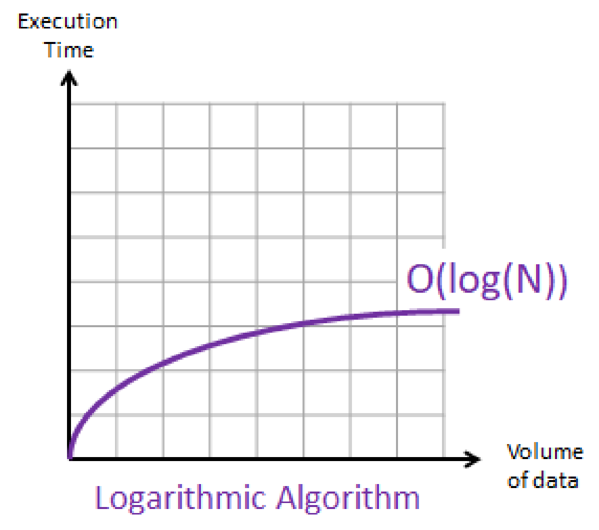
Constant - O(1)



Linear - O(n)

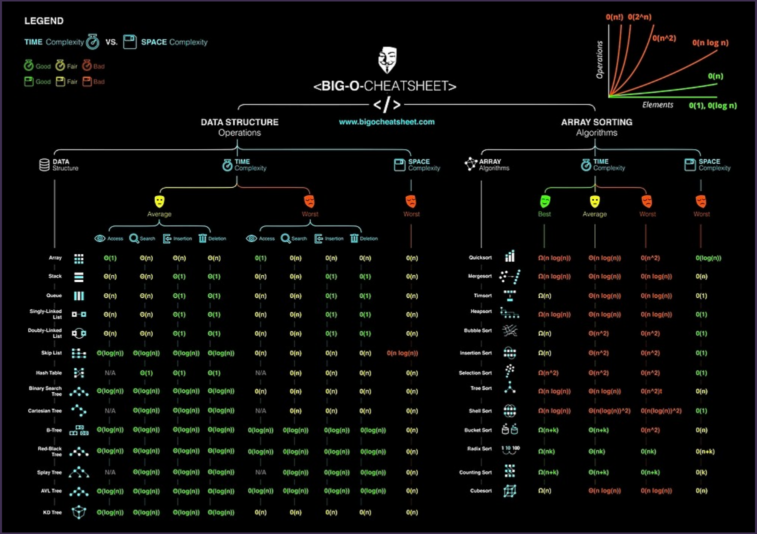


Logarithmic - O(log(n))



FALMOUTH UNIVERSITY

Big O Cheatsheet



Data Structure	Time Complexity: Average: Access	Average: Search	Average: Insertion	Average: Deletion	Worst: Access	Worst: Search	Worst: Insertion	Worst: Deletion	Space Complexity: Worst
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Algorithms	Time Complexity: Best	Average	Worst	Space Complexity: Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(1)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)t	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Dynamic Array

The Problem

The Problem

- Arrays in C# are fixed in size

The Problem

- Arrays in C# are fixed in size
- During development you need to know exactly how many item are going to be in the array

The Problem

- Arrays in C# are fixed in size
- During development you need to know exactly how many item are going to be in the array
- If you need to add elements and you don't have enough space, you will need to carry out the following

The Problem

- Arrays in C# are fixed in size
- During development you need to know exactly how many item are going to be in the array
- If you need to add elements and you don't have enough space, you will need to carry out the following
  - Create a new array of the appropriate size

The Problem

- Arrays in C# are fixed in size
- During development you need to know exactly how many item are going to be in the array
- If you need to add elements and you don't have enough space, you will need to carry out the following
  - Create a new array of the appropriate size
  - Copy elements from the old array into this new one

The Problem

- Arrays in C# are fixed in size
- During development you need to know exactly how many item are going to be in the array
- If you need to add elements and you don't have enough space, you will need to carry out the following
  - Create a new array of the appropriate size
  - Copy elements from the old array into this new one
  - Destroy the old array

The Problem

- Arrays in C# are fixed in size
- During development you need to know exactly how many item are going to be in the array
- If you need to add elements and you don't have enough space, you will need to carry out the following
  - Create a new array of the appropriate size
  - Copy elements from the old array into this new one
  - Destroy the old array
  - Add in the new element

The Problem

- Arrays in C# are fixed in size
- During development you need to know exactly how many item are going to be in the array
- If you need to add elements and you don't have enough space, you will need to carry out the following
  - Create a new array of the appropriate size
  - Copy elements from the old array into this new one
  - Destroy the old array
  - Add in the new element
- The above process can be quite costly

The Solution

The Solution

- Luckily in most programming languages we have a Data Structure which grows in size when we require it

The Solution

- Luckily in most programming languages we have a Data Structure which grows in size when we require it
  - In C# we have the **List** class

The Solution

- Luckily in most programming languages we have a Data Structure which grows in size when we require it
  - In C# we have the **List** class
- These classes have the same properties as an array

The Solution

- Luckily in most programming languages we have a Data Structure which grows in size when we require it
  - In C# we have the **List** class
- These classes have the same properties as an array
  - Items are located contiguously in memory

The Solution

- Luckily in most programming languages we have a Data Structure which grows in size when we require it
  - In C# we have the **List** class
- These classes have the same properties as an array
  - Items are located contiguously in memory
  - We can randomly access elements using an index

The Solution

- Luckily in most programming languages we have a Data Structure which grows in size when we require it
  - In C# we have the **List** class
- These classes have the same properties as an array
  - Items are located contiguously in memory
  - We can randomly access elements using an index
  - We can iterate through each element

The Solution

- Luckily in most programming languages we have a Data Structure which grows in size when we require it
  - In C# we have the **List** class
- These classes have the same properties as an array
  - Items are located contiguously in memory
  - We can randomly access elements using an index
  - We can iterate through each element
- You should consider using a Dynamic Array over a normal array

The Solution

- Luckily in most programming languages we have a Data Structure which grows in size when we require it
  - In C# we have the **List** class
- These classes have the same properties as an array
  - Items are located contiguously in memory
  - We can randomly access elements using an index
  - We can iterate through each element
- You should consider using a Dynamic Array over a normal array
- One caveat, Dynamic Arrays are slightly more expensive!

Use Case

Use Case

- Manage Enemies as they are spawned into the scene

Use Case

- Manage Enemies as they are spawned into the scene
- Keep track of players as they are added into the game

Use Case

- Manage Enemies as they are spawned into the scene
- Keep track of players as they are added into the game
- Inventory systems

C# List Example

```
List<int> scores=new List<int>();
scores.Add(100);
scores.Add(200);
foreach(int score in scores)
{
    Debug.Log("Score is "+score.ToString());
}
int player1Score=scores[0];
scores.Remove(100);
```

Additional Notes

Additional Notes

- Try to avoid insertion/deleting in the middle of the collection

Additional Notes

- Try to avoid insertion/deleting in the middle of the collection
- Searching the collection is linear and will increase as more elements are added (O(n))

Additional Notes

- Try to avoid insertion/deleting in the middle of the collection
- Searching the collection is linear and will increase as more elements are added (O(n))
- insertion/deleting at the end of the collection is constant in performance (O(1))

FALMOUTH
UNIVERSITY

Generic Types

Quick Aside - Generic Programming

Quick Aside - Generic Programming

- Generic Programming is where you write one piece of code which operates on many different types

Quick Aside - Generic Programming

- Generic Programming is where you write one piece of code which operates on many different types
- This uses a concept called Templates which act in proxy for the type

Quick Aside - Generic Programming

- Generic Programming is where you write one piece of code which operates on many different types
- This uses a concept called Templates which act in proxy for the type
- The Compiler then generates the code which uses the actual type

Look back at List

Look back at List

- In the previous section you would have noticed the following

Look back at List

- In the previous section you would have noticed the following
  - List<int>

Look back at List

- In the previous section you would have noticed the following
  - List<int>
- These are know as generic parameters and you should insert the data type that the collection will handle (including your own data types aka classes and structs)

Generic Programming

Generic Programming

- You can write your own generic classes and functions but this is beyond the scope of this class

Generic Programming

- You can write your own generic classes and functions but this is beyond the scope of this class
- C# examples - `http://www.tutorialsteacher.com/csharp/csharp-generics`

Generic Programming

- You can write your own generic classes and functions but this is beyond the scope of this class
- C# examples - `http://www.tutorialsteacher.com/csharp/csharp-generics`
- Word of warning, it is often difficult to write generic code

Generic Programming

- You can write your own generic classes and functions but this is beyond the scope of this class
- C# examples - `http://www.tutorialsteacher.com/csharp/csharp-generics`
- Word of warning, it is often difficult to write generic code
- If you have errors they are often difficult to isolate as the compiler messages are so cryptic

Linked List

The Problem

The Problem

- You have started using a dynamic array and you have noticed performance is poor on adding/removing

The Problem

- You have started using a dynamic array and you have noticed performance is poor on adding/removing
- You then realise that you are adding/removing elements from the middle of the collection

The Problem

- You have started using a dynamic array and you have noticed performance is poor on adding/removing
- You then realise that you are adding/removing elements from the middle of the collection
- You also realise that you don't require random access to elements in the collection

The Solution

The Solution

- In this case a Linked List would be a better choice

The Solution

- In this case a Linked List would be a better choice
  - In C# we have the **LinkedList** class

The Solution

- In this case a Linked List would be a better choice
  - In C# we have the **LinkedList** class
- Linked Lists contain elements (called Nodes) which usually have a reference to the previous and next Node in the list

The Solution

- In this case a Linked List would be a better choice
  - In C# we have the **LinkedList** class
- Linked Lists contain elements (called Nodes) which usually have a reference to the previous and next Node in the list
- This means that there is a slight increase in memory needed when working with lists

Use Case

Use Case

- If you AI character has to visit a series of waypoints, these could be stored in a list

Use Case

- If you AI character has to visit a series of waypoints, these could be stored in a list
- Your Player has a number of quests they can try and complete

Use Case

- If you AI character has to visit a series of waypoints, these could be stored in a list
- Your Player has a number of quests they can try and complete
- If the AI/Player carries an action and a number of systems need to be notified of the event

C# Linked List Example

```
LinkedList<Transform> waypoints=new LinkedList<
    Transform>();

waypoints.AddLast(GameObject.Find("Waypoint1").
    Transform);
waypoints.AddLast(GameObject.Find("Waypoint2").
    Transform);
waypoints.AddLast(GameObject.Find("Waypoint3").
    Transform);

foreach(Transfrom t in waypoints)
{
    Debug.Log("Waypoint Locations "+t.position.
        ToString());
}
```

C# Linked List Example

```
waypoints.AddFirst(GameObject.Find("Waypoint0").
    Transform);

LinkedListNode<Transform> waypoint2Node = linked.Find(
    GameObject.Find("Waypoint2"));
waypoints.AddAfter(waypoint2Node,GameObject.Find("
    SpecialQuest"));
```

Additional Notes

Additional Notes

- Linked Lists usually support constant time insertions and deletions in the collection ($O(1)$)

Additional Notes

- Linked Lists usually support constant time insertions and deletions in the collection (O(1))
- Also perform better than dynamic arrays for moving elements around the collection

Additional Notes

- Linked Lists usually support constant time insertions and deletions in the collection (O(1))
- Also perform better than dynamic arrays for moving elements around the collection
- This feature means that Linked Lists are a good data structure if you need to sort your data

Additional Notes

- Linked Lists usually support constant time insertions and deletions in the collection (O(1))
- Also perform better than dynamic arrays for moving elements around the collection
- This feature means that Linked Lists are a good data structure if you need to sort your data
- Main drawback of Linked Lists is that you can't have direct access to elements in the list, it takes linear time (O(n)) to access

Queue

The Problem

The Problem

- If you need to visit items in a certain (e.g front to back)

The Problem

- If you need to visit items in a certain (e.g front to back)
- Examples of this could be waypoints or commands to an AI character

The Solution

The Solution

- In this case a Queue would be a good choice

The Solution

- In this case a Queue would be a good choice
  - In C# we have the **Queue** class

The Solution

- In this case a Queue would be a good choice
  - In C# we have the **Queue** class
- This is **F**irst-**I**n-**L**ast-**O**ut data structure

The Solution

- In this case a Queue would be a good choice
  - In C# we have the **Queue** class
- This is **F**irst-**I**n-**L**ast-**O**ut data structure
- You add elements to the end of the queue and you remove elements from the start

Use Case

Use Case

- An RTS game where you can add orders to a unit, these are then carried out sequence

Use Case

- An RTS game where you can add orders to a unit, these are then carried out sequence
- An RTS where you have a base which produces units

Use Case

- An RTS game where you can add orders to a unit, these are then carried out sequence
- An RTS where you have a base which produces units
- A spawning system, where you have to defeat enemies in a specific order

C# Queue Example

```
Queue<GameObject> unitsToBuild=new Queue<GameObject>()
    ;

unitsToBuild.Enqeue(soliderPrefab);
unitsToBuild.Enqeue(builderPrefab);
unitsToBuild.Enqeue(tankPrefab);

foreach(GameObject go in unitsToBuild)
{
    Debug.Log("Units to build "+go.name);
}
```

C# Queue Example

```
GameObject nextUnitToBuild=unitsToBuild.Peek();

unitsToBuild.Dequeue();
```

FALMOUTH UNIVERSITY

Stack

The Problem

The Problem

- If you need to manage the state of an AI character

The Problem

- If you need to manage the state of an AI character
- If you need to implement a Undo system

The Solution

The Solution

- A Stack would be a good choice

The Solution

- A Stack would be a good choice
  - In C# we have the **Stack** class

The Solution

- A Stack would be a good choice
  - In C# we have the **Stack** class
- This is **L**ast-**I**n-**F**irst-**O**ut data structure

The Solution

- A Stack would be a good choice
  - In C# we have the **Stack** class
- This is **L**ast-**I**n-**F**irst-**O**ut data structure
- You add elements to the top of the stack and you remove elements from the top

C# Stack Example

```
Stack<Command> issuedCommands=new Stack<Command>();

issuedCommands.Push(new Command("Edit"));
issuedCommands.Push(new Command("Create"));
issuedCommands.Push(new Command("Update"));
```

C# Stack Example

```
Command lastCommandIssued=issuedCommands.Peek();

Command lastCommandIssued=issuedCommands.Pop();
```

FALMOUTH UNIVERSITY

Associative Array: Dictionary

The Problem

The Problem

- If you need to store one unique copy of an element

The Problem

- If you need to store one unique copy of an element
- You want to access an element via a key

The Problem

- If you need to store one unique copy of an element
- You want to access an element via a key
- You are doing lots of searches for an element

The Solution

The Solution

- You should use an Associative array

The Solution

- You should use an Associative array
  - In C# we have the **Dictionary** class

The Solution

- You should use an Associative array
  - In C# we have the **Dictionary** class
- These data structures are structured as key-value pair

The Solution

- You should use an Associative array
  - In C# we have the **Dictionary** class
- These data structures are structured as key-value pair
- It allows you to retrieve the items via the key

The Solution

- You should use an Associative array
  - In C# we have the **Dictionary** class
- These data structures are structured as key-value pair
- It allows you to retrieve the items via the key
- This makes it a good choice for looking up large data sets

Use Case

Use Case

- If you are creating a resource management system for handling textures, models or other assets

Use Case

- If you are creating a resource management system for handling textures, models or other assets
- Localisation system, each language is stored in an Associative Array

Use Case

- If you are creating a resource management system for handling textures, models or other assets
- Localisation system, each language is stored in an Associative Array
- Unit Manager, a class to manage units created in the game

Use Case

- If you are creating a resource management system for handling textures, models or other assets
- Localisation system, each language is stored in an Associative Array
- Unit Manager, a class to manage units created in the game
- Save Game System

C# Dictionary Example

```
Dictionary<string,int> highScoreTable=new Dictionary< ↩
    string,int>();

highScores.Add("Brian",200);
highScores.Add("Sarah",2000);
highScores[Julia]=4000;

foreach(KeyValuePair<string,int> pair in  ↩
    highScoreTable)
{
    Debug.Log("High Score "+pair.Key+" "+pair.Value);
}
```

C# Dictionary Example

```
if (highScores.ContainsKey("Brian"))
{
    int score=highScores["Brian"];
}

highScores.Remove("Sarah");
```

Additional Notes

Additional Notes

- Iterating over a map has a slightly annoying syntax

Additional Notes

- Iterating over a map has a slightly annoying syntax
- Associative Arrays tend to have good performance for retrieval (O (log n))

Additional Notes

- Iterating over a map has a slightly annoying syntax
- Associative Arrays tend to have good performance for retrieval (O (log n))
- If you add an item and its key already exists it may overwrite the value

Operations on collections

Sorting

Sorting

- Sorting is where we order the items in a collection in a specific order

Sorting

- Sorting is where we order the items in a collection in a specific order
- There are a whole bunch of sorting algorithms including; Insertion sort, Heap sort, Quick sort (please read about these!)

Sorting

- Sorting is where we order the items in a collection in a specific order
- There are a whole bunch of sorting algorithms including; Insertion sort, Heap sort, Quick sort (please read about these!)
- In C#, the best sorting algorithm will be picked depending on the size of the collection

Sorting

- Sorting is where we order the items in a collection in a specific order
- There are a whole bunch of sorting algorithms including; Insertion sort, Heap sort, Quick sort (please read about these!)
- In C#, the best sorting algorithm will be picked depending on the size of the collection
- In C++, this depends on the compiler implementation

Sorting

- Sorting is where we order the items in a collection in a specific order
- There are a whole bunch of sorting algorithms including; Insertion sort, Heap sort, Quick sort (please read about these!)
- In C#, the best sorting algorithm will be picked depending on the size of the collection
- In C++, this depends on the compiler implementation
- Most of the common data types don't need additional work

Sorting

- Sorting is where we order the items in a collection in a specific order
- There are a whole bunch of sorting algorithms including; Insertion sort, Heap sort, Quick sort (please read about these!)
- In C#, the best sorting algorithm will be picked depending on the size of the collection
- Most of the common data types don't need additional work
- For custom classes, we have to write our own sorting algorithm

Sorting C#

Sorting C#

- There are few ways to sort a collection

Sorting C#

- There are few ways to sort a collection
  1. Provide a custom delegate function for the sort

Sorting C#

- There are few ways to sort a collection
    1. Provide a custom delegate function for the sort
    2. Provide a custom class which inherits from **IComparer**

Sorting C#

- There are few ways to sort a collection
    1. Provide a custom delegate function for the sort
    2. Provide a custom class which inherits from **IComparer**
    3. Your own class has to inherit from **IComparable**

Sorting C#

- There are few ways to sort a collection
  1. Provide a custom delegate function for the sort
  2. Provide a custom class which inherits from **IComparer**
  3. Your own class has to inherit from **IComparable**
- Often you will use option 3 as the default sort

Sorting C#

- There are few ways to sort a collection
  1. Provide a custom delegate function for the sort
  2. Provide a custom class which inherits from **IComparer**
  3. Your own class has to inherit from **IComparable**
- Often you will use option 3 as the default sort
- Which then be override by option 1

C# Example - Sorting with Delegate

```
struct Character
{
    string name;
    int health;
    int strength;
}

//Adding omitted!
List<Character> characters=new List<Character>();

//Sort by health
characters.Sort(delegate (Character c1, Character c2)
{
    return (c1.health.CompareTo(c2.health));
});
```

C# Example - Sorting with ICompareable

```
struct Character:IComparable<Character>
{
string name;
int health;
int strength;

// sort by name
 public int CompareTo(Character compareCharacter)
 {
     return name.CompareTo(comareCharacter.name);
 }
}

//Adding omitted!
List<Character> characters=new List<Character>();

//Sort will use the CompareTo in the struct or class
characters.Sort()
```

C# - Points to note

C# - Points to note

- The **CompareTo** function returns an int which can be the following

C# - Points to note

- The **CompareTo** function returns an int which can be the following
  - Less than zero: The instance precedes the one passed in

C# - Points to note

- The **CompareTo** function returns an int which can be the following
  - Less than zero: The instance precedes the one passed in
  - Zero: The objects are in the same order

C# - Points to note

- The **CompareTo** function returns an int which can be the following
  - Less than zero: The instance precedes the one passed in
  - Zero: The objects are in the same order
  - Greater than zero: The instance follows the one passed in

Exercise

Exercise 1 - Collections

1. Download one of the following projects as a zip file
   - BA Students - `https://github.com/Falmouth-Games-Academy/GAM160-Exercises`
   - BSc Students - `https://github.com/Falmouth-Games-Academy/COMP140-Exercises`
2. Add additional items to the collection
3. Display these to the screen

Exercise 2 - Sorting

1. Write a default sort, so that the items are sorted by name
2. Sort the collection when the **s** key is pressed
3. Write another sort, to sort by score, trigger this off by a key press
4. Write another sort, to sort by age, trigger this off by a key press

FALMOUTH UNIVERSITY

Exercise 3 - Searching

1. Investigate how to search for items in collections
2. Add code to search for specific items in the collections
3. Add visual representation to show that the search has completed, this could be a colour change or just displaying the found item elsewhere on the screen

References

https://unity3d.com/learn/tutorials/modules/intermediate/scripting/lists-and-dictionaries
https://www.101computing.net/big-o-notation/
http://bigocheatsheet.com/