



Data Structures & Algorithms

Instructor: Phan Thi Thanh Tra

Student: Nguyen Van Hieu

Class: GCD0819

A decorative banner consisting of a central rectangular box with rounded corners, flanked by two stylized ribbon-like shapes that point outwards.

Da Nang, 2019

Summary

In Assignment 2, it will assign:

- Design ADT / algorithms for these 2 structures and implement a demo version with message is a string of maximum 250 characters. The demo should demonstrate some important operations of these structures. Even it's a demo, errors should be handled carefully by exceptions and some tests should be executed to prove the correctness of algorithms / operations.
- Write a report of the implementation of the 2 data structures
- How to measure the efficiency of related algorithms
- Evaluate the use of ADT in design and development, including the complexity, the trade-off and the benefits.

P4 Implement a complex ADT and algorithms in an executable programming language to solve a well-defined problem.

Firstly, define Stack as:

THE STACK ADT

- A Stack is a collection of objects inserted and removed according to the Last In First Out (LIFO) principle. Think of a stack of dishes.
- Push and Pop are the two main operations

STACK ADT OPERATIONS

- **push(o)**: Insert o at top of stack
 - Input: Object; Output: None
- **pop()**: Remove top object; error if empty
 - Input: None; Output: Object removed
- **size()**: Return number of objects in stack
 - Input: None; Output: Integer
- **isEmpty()**: Return a boolean indicating stack empty
 - Input: None; Output: Boolean
- **top()**: Return top object without removing; error if empty
 - Input: None; Output: Object

EXAMPLE

```
Stack MyStack4 = new Stack ();  
  
MyStack.Push ("Me");  
MyStack.Push ("It's");  
MyStack.Push ("HieuNguyen");  
  
Console.WriteLine (" Have elements: {0}", MyStack.Count);  
Console.WriteLine (" Elements of Stack as: {0}", MyStack.Peek());  
Console.WriteLine (" Elements of Stack after call Peek function: {0}", MyStack.Count);  
Console.WriteLine (" Popping...");  
  
int Length = MyStack.Count;  
for (int i = 0; i < Length; i++)  
{ Console.Write (" " + MyStack.Pop());}  
  
Console.WriteLine ();  
  
Console.WriteLine ("Elements of Stack after call Pop|function: {0}", MyStack.Count);
```

Annotations in the image:

- Create a Stack empty**: Points to `Stack MyStack4 = new Stack ();`
- Add some elements to the Queue via the Enqueue function.**: Points to the three `MyStack.Push` lines.
- Export the command line**: Points to the `Console.WriteLine` line after the popping loop.

Secondly, define Queue as:

THE QUEUE ADT

- Recall the waiting list for courses during registration for courses? When a seat opens up, the first one who joined the waiting list is the first to get a chance to add to the course.
- This is a queue, works on the first in first out principle. Two access point: front and rear
- Other examples are: Call centers, printer queue, etc

QUEUE ADT OPERATIONS

- **enqueue(o)**: Insert o at rear of queue
 - Input: Object; Output: None
- **dequeue()**: Remove object at front; error if empty
 - Input: None; Output: Object removed
- **size()**: Return number of objects in queue
 - Input: None; Output: Integer
- **isEmpty()**: Return a boolean indicating queue empty
 - Input: None; Output: Boolean
- **first()**: Return object at front without removing; error if empty
 - Input: None; Output: Object

EXAMPLE

```
Queue MyQueue = new Queue ();  
MyQueue.Enqueue ("HieuNguyen");  
MyQueue.Enqueue ("It's");  
MyQueue.Enqueue ("Me");  
Console.WriteLine (" Have elements: {0}", MyQueue4.Count);  
Console.WriteLine (" First element of Queue as: {0}", MyQueue4.Peek ());  
Console.WriteLine ("Elements of Queue after call Peek function: {0}", MyQueue4.Count);  
Console.WriteLine (" Popping...");  
int Length = MyQueue4.Count;  
for (int i = 0; i < Length; i++)  
{ Console.Write (" " + MyQueue.Dequeue ());}  
Console.WriteLine ();  
Console.WriteLine (" Elements of Queue after call Pop function {0}", MyQueue4.Count);
```

Create a Queue empty

Add some elements to the Queue via the Enqueue function.

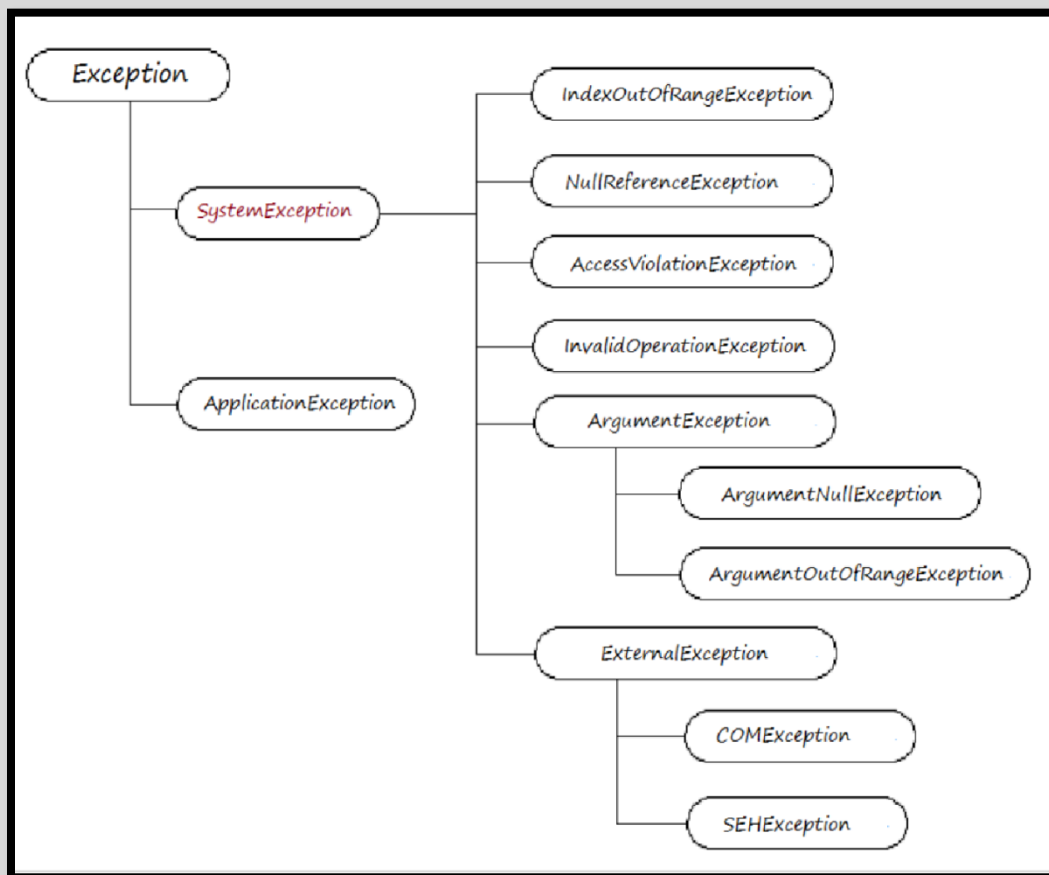
Export the command line

P5. Implement error handling and report test results

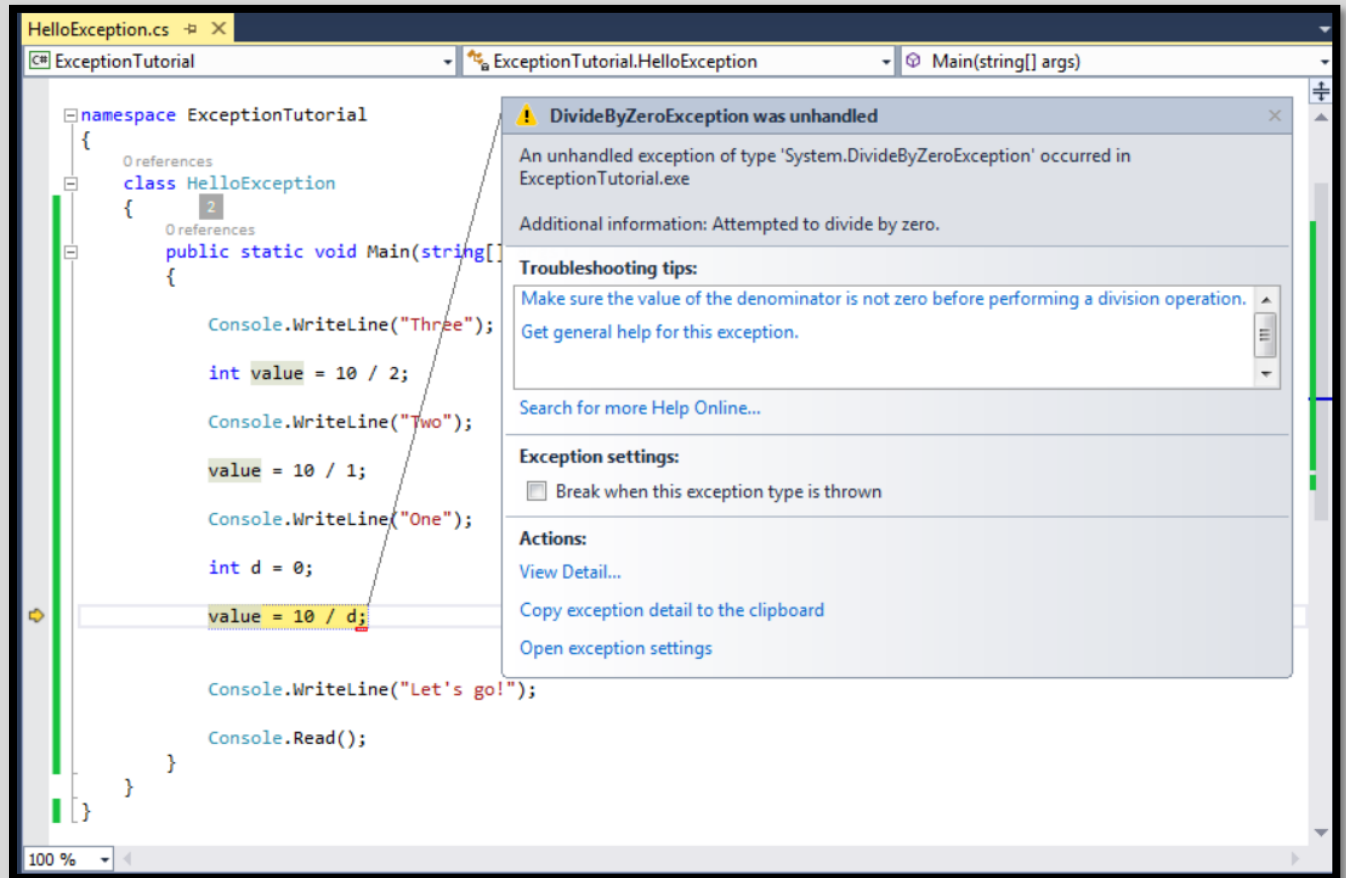
Exception handle?

Exception handling is the process of responding to the occurrence, during computation, of exceptions – anomalous or exceptional conditions requiring special processing – often disrupting the normal flow of program execution. It is provided by specialized programming language constructs, computer hardware mechanisms like interrupts or operating system IPC facilities like signals.

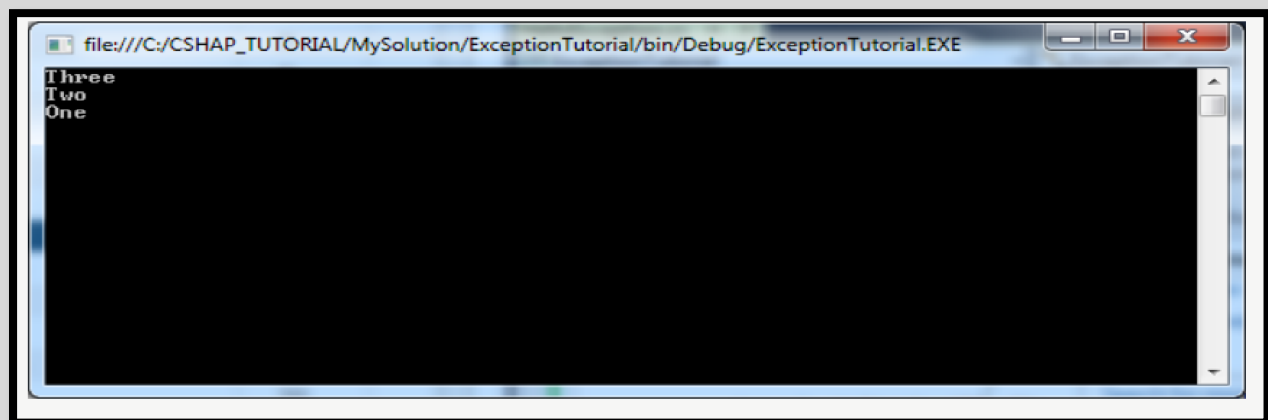
And below here is some “Exceptions” popular are available in C#



Firstly, create an example of an error code, which is caused by division by 0



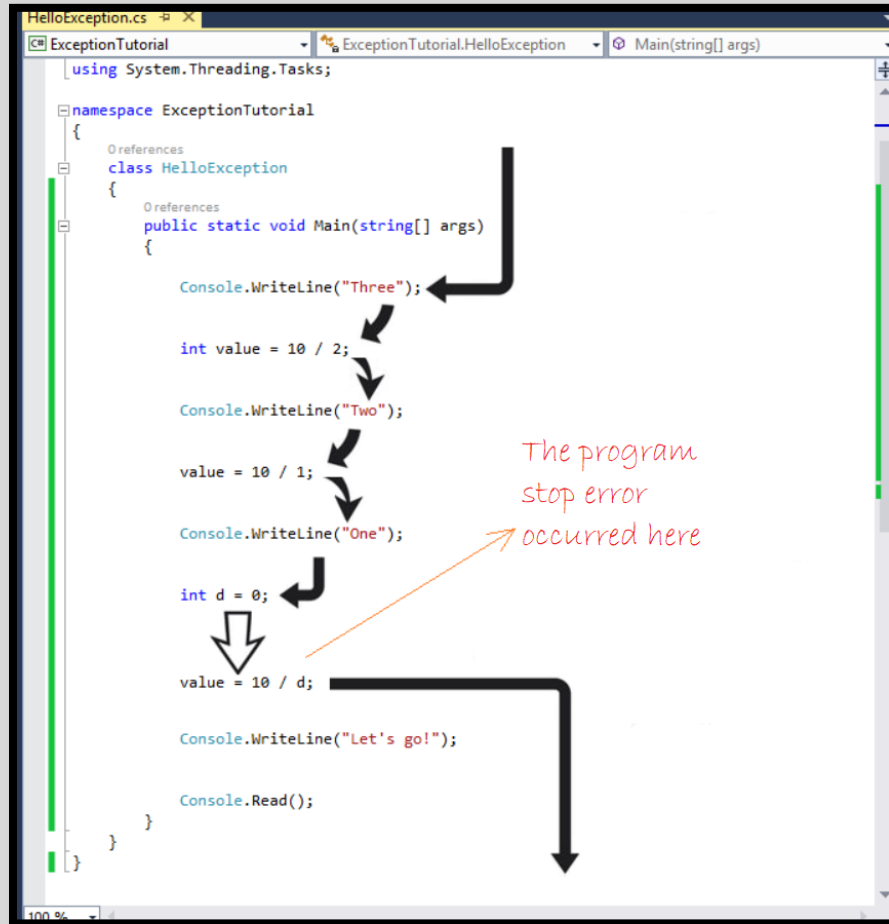
An example of an error code



Result when run program

Then, please see the running direction of the program:

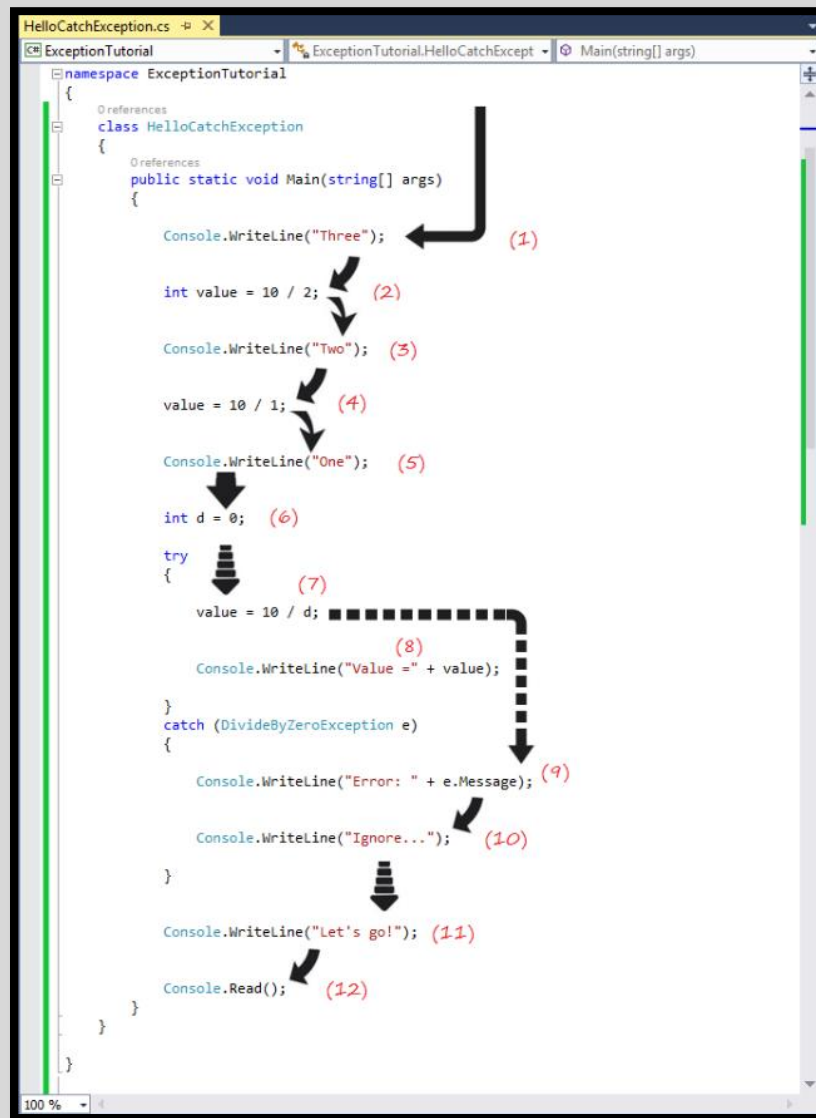
- Programming runs completely normal from the first step to step (5)
- Step (6) has a trouble which is caused by division by 0
- And Program stopped and didn't perform step (7)



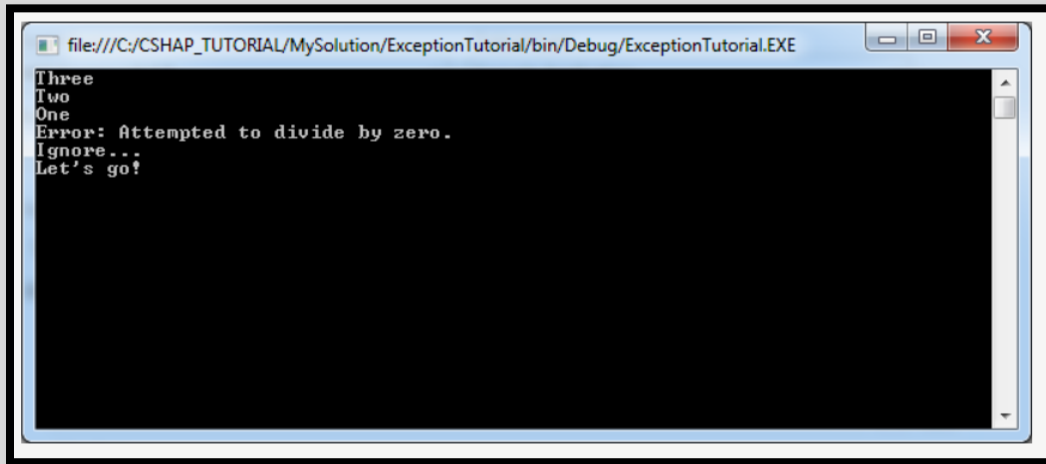
And we'll repair above this code:

I'll explain to clear more about this code in below:

- Step (1)-(6) which completely normal
- Exception occurred in step (7), problem divided by 0
- Instantly, which execute command code "catch", step (8) be skip
- Step (9) (10) be execute
- Step (11) (12) be excute



And result



```
file:///C:/CSHAP_TUTORIAL/MySolution/ExceptionTutorial/bin/Debug/ExceptionTutorial.EXE
Three
Two
One
Error: Attempted to divide by zero.
Ignore...
Let's go!
```

P6 Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm

What's Asymptotic analysis?

- Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.
- Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.
- Usually, the time required by an algorithm falls under three types –
 - ✓ Best Case – Minimum time required for program execution.
 - ✓ Average Case – Average time required for program execution.
 - ✓ Worst Case – Maximum time required for program execution.

The purpose of asymptotic analysis

- ✓ To estimate how long a program will run.
- ✓ To estimate the largest input that can reasonably be given to the program.
- ✓ To compare the efficiency of different algorithms.
- ✓ To help focus on the parts of code that are executed the largest number of times.
- ✓ To choose an algorithm for an application.

In additional, for each input sizes:

❖ Large input sizes:

Algorithms with well complexity are usually more complicated. This can increase coding time and the constants.

❖ Small input sizes:

Asymptotic analysis ignores small input sizes. At small input sizes, constant factors or low order terms could dominate running time, causing Y to outperform X.

Overview

Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take $1000n\log n$ and $2n\log n$ time respectively on a machine. Both of these algorithms are asymptotically same (order of growth is $n\log n$). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis.

Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

(Sudhanshu Ratnaparkhee, Technical Leader at KPIT Technologies GmbH (2019-present))

P7. Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example.




Algorithm efficiency

Two areas are important for performance:

➤ Space efficiency - the memory required, also called, space complexity

There are some circumstances where the space/memory used must be analyzed. For example, for large quantities of data or for embedded systems programming.

Components of space/memory use:

-  **Instruction space** Affected by: the compiler, compiler options, target computer (cpu)
-  **Data space** Affected by: the data size/dynamically allocated memory, static program variables,
-  **Run-time stack space** Affected by: the compiler, run-time function calls and recursion, local variables, parameters

The space requirement has fixed/static/compile time and a variable/dynamic/runtime components. Fixed components are the machine language instructions and static variables. Variable components are the runtime stack usage and dynamically allocated memory usage.





One circumstance to be aware of is, does the approach require the data to be duplicated in memory (as does merge sort). If so we have $2N$ memory use.

Space efficiency is something we will try to maintain a general awareness of.

➤ Time efficiency - the time required, also called time complexity

Clearly the more quickly a program/function accomplishes its task the better.

The actual running time depends on many factors:

-  The speed of the computer: cpu (not just clock speed), I/O, etc.
-  The compiler, compiler options .
-  The quantity of data - ex. search a long list or short.
-  The actual data - ex. in the sequential search if the name is first or last.

Comparison illustrating

In instance, these are two algorithm results we are comparing, The QuickSort and TimSort algorithms are both fast although QuickSort which relative simplicity and smaller space requirements.

TimSort

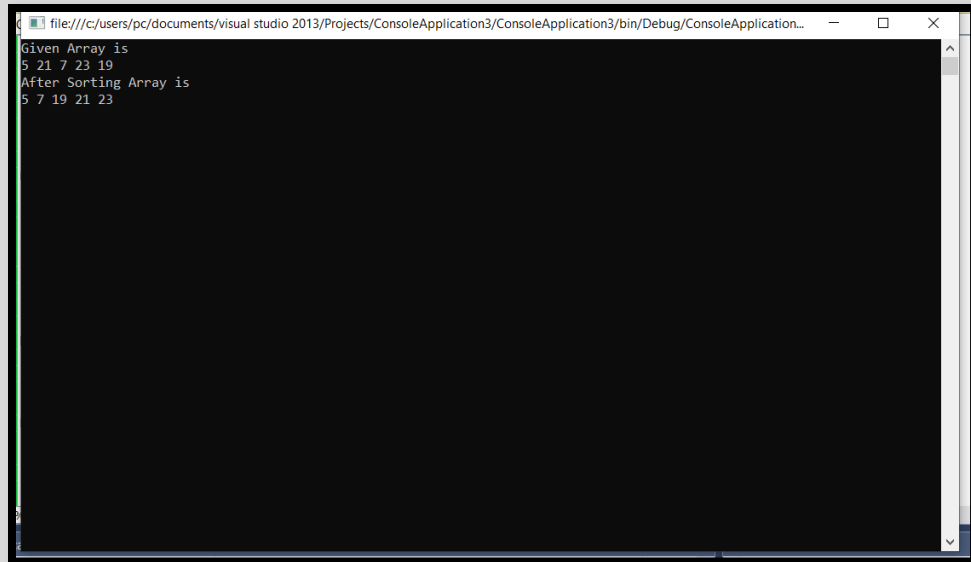
```
class GFG
{
    public const int RUN = 32;
    public static void insertionSort(int[] arr, int left, int right)
    {
        for (int i = left + 1; i <= right; i++)
        {
            int temp = arr[i];
            int j = i - 1;
            while (arr[j] > temp && j >= left)
            {
                arr[j+1] = arr[j];
                j--;
            }
            arr[j+1] = temp;
        }
    }
    public static void merge(int[] arr, int l, int m, int r)
    {
        int len1 = m - l + 1, len2 = r - m;
        int[] left = new int[len1];
        int[] right = new int[len2];
        for (int x = 0; x < len1; x++)
            left[x] = arr[l + x];
        for (int x = 0; x < len2; x++)
            right[x] = arr[m + 1 + x];

        int i = 0;
        int j = 0;
        int k = l;
        while (i < len1 && j < len2)
        {
            if (left[i] <= right[j])
            {
                arr[k] = left[i];
                i++;
            }
            else
            {
                arr[k] = right[j];
                j++;
            }
            k++;
        }
        while (i < len1)
            arr[k++] = left[i++];
        while (j < len2)
            arr[k++] = right[j++];
    }
}
```

Figure 1: code TimSort in C#

```
        }
        k++;
    }
    while (i < len1)
    {
        arr[k] = left[i];
        k++;
        i++;
    }
    while (j < len2)
    {
        arr[k] = right[j];
        k++;
        j++;
    }
}
public static void timSort(int[] arr, int n)
{
    for (int i = 0; i < n; i += RUN)
        insertionSort(arr, i, Math.Min((i+31), (n-1)));
    for (int size = RUN; size < n; size = 2*size)
    {
        for (int left = 0; left < n; left += 2*size)
        {
            int mid = left + size - 1;
            int right = Math.Min((left + 2*size - 1), (n-1));
            merge(arr, left, mid, right);
        }
    }
    public static void printArray(int[] arr, int n)
    {
        for (int i = 0; i < n; i++)
            Console.Write(arr[i] + " ");
        Console.WriteLine();
    }
    public static void Main()
    {
        int[] arr = {5, 21, 7, 23, 19};
        int n = arr.Length;
        Console.WriteLine("Given Array is\n");
        printArray(arr, n);
        timSort(arr, n);
        Console.WriteLine("After Sorting Array is\n");
        printArray(arr, n);
    }
}
```

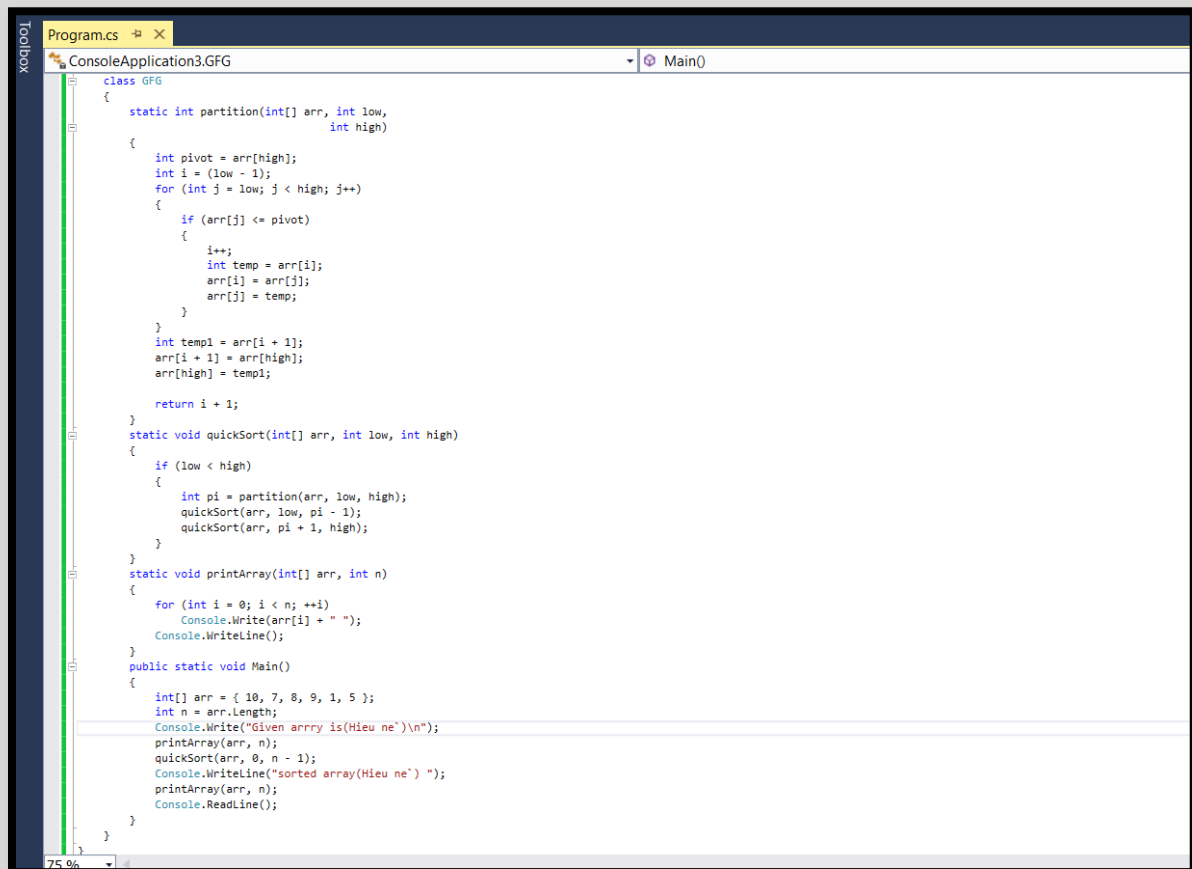
Figure 2: continue code TimSort in C#



```
file:///c:/users/pc/documents/visual studio 2013/Projects/ConsoleApplication3/ConsoleApplication3/bin/Debug/ConsoleApplication...
Given Array is
5 21 7 23 19
After Sorting Array is
5 7 19 21 23
```

Result of code TimSort in C#

QuickSort



```
Program.cs
ConsoleApplication3.GFG
Main()

class GFG
{
    static int partition(int[] arr, int low,
                        int high)
    {
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j < high; j++)
        {
            if (arr[j] <= pivot)
            {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp1 = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp1;

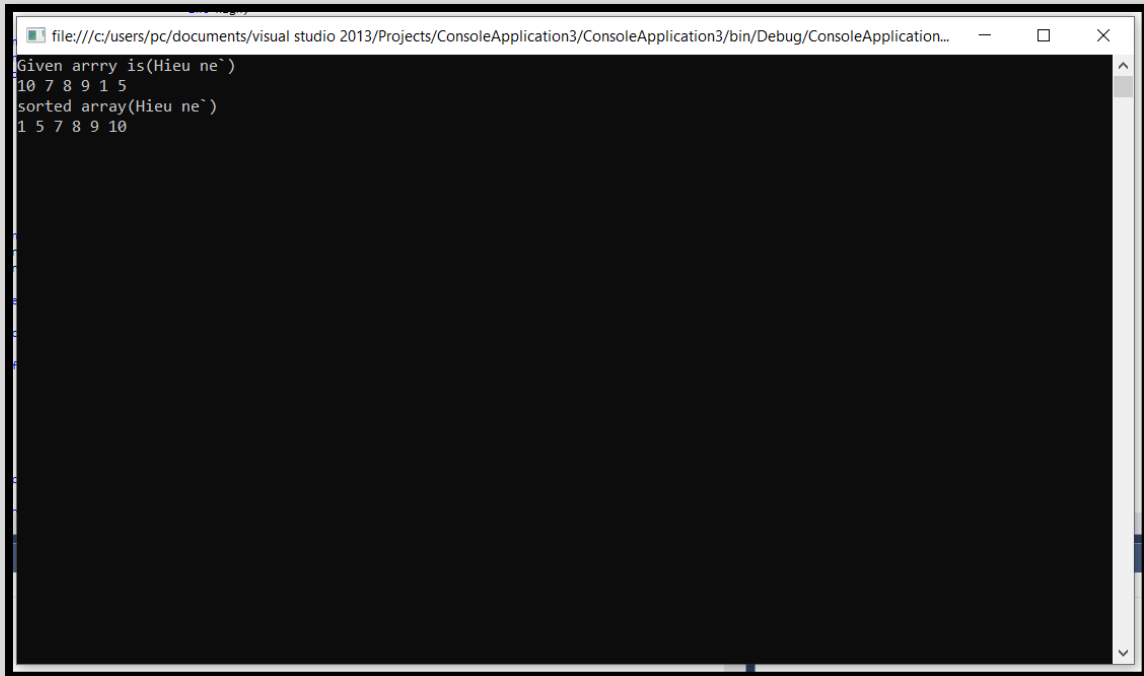
        return i + 1;
    }

    static void quickSort(int[] arr, int low, int high)
    {
        if (low < high)
        {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

    static void printArray(int[] arr, int n)
    {
        for (int i = 0; i < n; ++i)
            Console.Write(arr[i] + " ");
        Console.WriteLine();
    }

    public static void Main()
    {
        int[] arr = { 10, 7, 8, 9, 1, 5 };
        int n = arr.Length;
        Console.WriteLine("Given array is(Hieu ne')\n");
        printArray(arr, n);
        quickSort(arr, 0, n - 1);
        Console.WriteLine("sorted array(Hieu ne') ");
        printArray(arr, n);
        Console.ReadLine();
    }
}
```

Figure1: code QuickSort in C#



```
file:///c:/users/pc/documents/visual studio 2013/Projects/ConsoleApplication3/ConsoleApplication3/bin/Debug/ConsoleApplication...
Given array is(Hieu ne`)
10 7 8 9 1 5
sorted array(Hieu ne`)
1 5 7 8 9 10
```

Figure2: result QuickSort in C#

M4. Demonstrate how the implementation of an ADT/algorithm solves a well-defined problem

What's Middleware?

Middleware is the software that connects network-based requests generated by a client to the back-end data the client is requesting. It is a general term for software that serves to "glue together" separate, often complex and already existing programs.

How Middleware works?

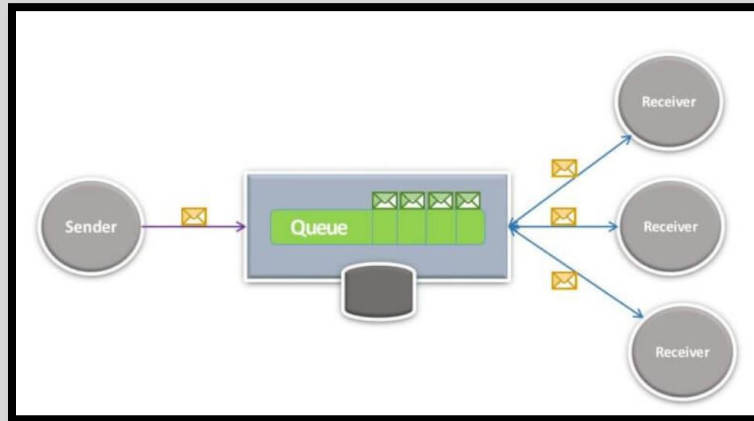
All network-based requests are essentially attempts to interact with back-end data. That data might be something as simple as an image to display or a video to play, or it could be as complex as a history of banking transactions.

The requested data can take on many different forms and may be stored in a variety of ways, such as coming from a file server, fetched from a message queue or persisted in a database. The role of middleware is to enable and ease access to those back-end resources.

Why using Message Queuing?

Firstly, **what message queuing?** - A message queue is a form of asynchronous service-to-service communication used in serverless and microservices architectures. Messages are stored on the queue until they are processed and deleted. Each message is processed only once, by a single consumer. Message queues can be used to decouple heavyweight processing, to buffer or batch work, and to smooth spiky workloads.

Why use message queuing and Reasons to USE MESSAGE QUEUING?



REDUNDANCY VIA PERSISTENCE

Queues help with redundancy by making the process that reads the message confirm that it completed the transaction and it is safe to remove it. If anything fails, worst case scenario, the message is persisted to storage somewhere and won't be lost. It can be reprocessed later.

IMPROVE WEB APPLICATION PAGE LOAD TIMES

Queues can be useful in web applications to do complex logic in a background thread so the request can finish for the user quickly.

MONITORING

Message queuing systems enable you to monitor how many items are in a queue, the rate of processing messages, and other stats. This can be very helpful from an application monitoring standpoint to keep an eye on how data is flowing through your system and if it is getting backed up.

Message Oriented Middleware

OVERVIEW

- MOM provides a clean method of communication between disparate software applications and emerged as a approach that distributed enterprise systems are built

- The client of MOM system can send and receive messages from other clients of the messaging system
- Each client connects to one or more servers that act as an intermediary while sending and receiving messages
- The MOM platforms allows enterprises to build cohesive system

MESSAGE-ORIENTED MIDDLEWARE – ADVANTAGES

Asynchronous Messaging: MOM comprises a category of inter-application communication software that usually relies on asynchronous message-passing, as opposed to request-response architecture. In case of asynchronous systems, message queues provide temporary storage when the destination program is busy or unable to get connected.

MESSAGE-ORIENTED MIDDLEWARE – DISADVANTAGES

MOM requires an extra component in the architecture, the message transfer agent (message broker). As with any system, adding another component can lead to reductions in performance and make the system as a whole more difficult and expensive to maintain. When introduced due to lack of standards governing the use of message-oriented middleware has caused problems.

IMPLEMENTATION

- The message queue is a fundamental concept within MOM
- Queues provide the ability to store messages on a MOM platform
- MOM clients are able to send and receive messages to and from a queue
- Queues are central to the implementation of the asynchronous interaction model within MOM
- Usually the messages contained within a Queue is sorted in a particular order
- The standard queue found in a messaging system is the First-In First-Out (FIFO) queue
- Many attributes like queue's name, queue's size, the save threshold of the queue, message-sorting algorithm etc. can be configured
- MOM platforms support multiple queue types for different purpose

M5. Interpret what a trade-off is when specifying an ADT using an example to support your answer

- Time Space Trade-Off of algorithms. ... The best algorithm, hence best program to solve a given problem is one that requires less space in memory and takes less time to execute its instruction or to generate output
- The best algorithm, hence best program to solve a given problem is one that requires less space in memory and takes less time to execute its instruction or to generate output. But in practice, it is not always possible to achieve both of these objectives. As said earlier, there may be more than one approaches to solve a same problem. One such approach may require more space but takes less time to complete its execution. Thus we may have to sacrifice one at the cost of the other. That is what we can say that there exists a time space trade off among algorithms.
- Therefore, if space is our constraints then we have to choose a program that requires less space at the cost of more execution time. Other than that, if time is our constraint, then we have to choose a program that takes less time to complete its execution of statements at the cost of more space.
- Once running a program there are two important principles to know the efficiency of the code.
 1. Memory it needs to execute
 2. Time for which it need CPU resources
- A space-time or time-memory tradeoff is a way of solving a problem or calculation in less time by using more storage space (or memory), or by solving a problem in very little space by spending a long time.
- To measure Space and Time Complexity we use Asymptotic Analysis. Usually, we do not mess with how many particular seconds particular algorithm takes. We calculate time and space complexity in the form of mathematical function.