Mastering
The
Technical Interview

Know your technical interview questions
and ace the interview

Mastering
The
Technical Interview

TREVANNON
BURGESS

Be prepared

And succeed

Utopia Now!
by Trevannon Burgess

First Edition

Published by TrevyCo Publishing

Printed in Seattle

10 9 8 7 6 5 4 3 2 1

Dedicated to my parents…
They always believed in me –
even when they were scolding me.

Table of Contents

An interview is like a minefield.
-- Michelle Williams --

The Technical Interview

It’s not about how much you know

But how much you can impress

1. INTRODUCTION

You know you're in trouble when they ask,
'Why do you want to work for us?'

In my opinion, the technical interview is one of the greatest challenges a software developer will face in their professional career.

The good/bad news is that they always ask the same questions, regardless of your level of expertize.

In this book I will share with you my experiences dealing with hiring managers and typical areas to look out for.

Be aware that they can and will ask stupid questions.

> Canadate: I created C# chess game with AI.
>
> Manager: That's great. I love your game. Do you know what a C# class is?

Or this:

> Canadate: I've been working as an SDET contractor at Microsoft for the last six years.
>
> Manager: That's Great. Have you ever used Visual Studios?

I was recently asked that question at a Microsoft interview. This was asked despite the fact that my resume said I used Visual Studios in several of my projects. Also, since I've been contracting at Microsoft for several years, the answer should be self-evident to the manager.

Intended Audience

This book is intended for software developers facing the interview process.

How this book is organized

This book will focus on the main themes a hiring manager looks for, namely:

- Object Oriented Programming concepts
- Design Patterns

- Data Structures and Algorithms
- Testing and Debugging
- Questions on previous projects

Suggestions to Recruiters/Hiring Managers

In my opinion, the technical interview is useless in accessing a person's technical ability.

In my opinion, the best way to get high quality professionals to work for you is the *contract to hire* route.

You bring someone in on a contract basis. You then evaluate that person for the next several months, to see if they have the skills you are looking for and if they fit in with the corporate environment.

This way, you see the actual potential a person has and if they are a good fit for your company. Also, since they were hired on a contractual basis, it's easy to let they go should they not work out.

Also, I recommend the book, *How to be a STAR at work,* by Robert. E. Kelley. This book teaches anyone how to become a high-performance worker.

2. OBJECT ORIENTED PROGRAMMING CONCEPTS

The best situation to be in is
when they want
you even before they interview you.

Knowledge of Object Oriented concepts is essential in the technical interview. They will always ask you questions such as, 'What is a class'. It doesn't matter how many years of experience you have.

This is a pitfall for senior level professions. The reason is that fundamental principles have already been assimilated. The professional uses them without thinking about them. As a result they can fall into traps a high school graduate would never fall into.

The answer is to review the basics before the interview.

2.1. Basic OOP

Basic concepts are programming principles that all computer science students should know and veterans tend to forget. (See: C# Programming Guide)

Class

Definition: A class is a basic construct in Object Oriented Programming, which supports Encapsulation, Inheritance, and Polymorphism.

It is the responsibility of the class to ensure that its states are externally consistent.

A class may be in an inconsistent state while a method is performing an operation. However, once the operation is complete, the state must once again be consistent.

1. Encapsulation

Encapsulation allows you to hide data and implementation details and expose them in a controlled manner through publically exposed methods.

2. Inheritance

Inheritance allows you to extend a parent class with new functionality, allowing you to override parent methods and add new child methods.

3. Polymorphism

Polymorphism occurs when an instance of a child class is treated as an instance of a parent class.

Calling a method on the parent class will then express different behaviors, based on how the child class has overridden the parent class method.

Method

Methods are used to encapsulate logical operations.

Methods are not intended for code reuse. That is a side benefit.

The purpose of methods is to encapsulate actions performed of input data. When creating methods, you must ask yourself – What are the logical operations you are exposing through your API?

Properties

In traditional OOP properties are just methods.

Properties expose the state of an object in a controlled manner. It is the responsibility of properties to ensure that the object's state is always externally consistent.

Note: An object must never expose internal fields. Internal states must only be exposed through managed properties or methods.

External Data Consistency

An object has fields that has both allowed and forbidden states. It is the job of the object to make sure its internal fields are in a consistent state.

The internal state of an object may be in a forbidden state while a method is performing an operation.

Objects

Objects are self-describing things with internal structure not visible to the external world. They encompass one single idea or concept and they provide methods that allow external objects to interact with them.

Virtual Methods

A virtual method is a method that is intended to be overridden in a derived class, and is required for polymorphic behavior. If it not overridden, then the base class method is used when the method is called using polymorphism.

Abstract Methods

Abstract methods are placeholders for missing functionality. These methods must be defined in child classes before object can be created.

Sealed Class (C#)

A sealed class cannot be inherited.

A method can be sealed, but only if it is overriding a base method.

2.2. SOLID

1. **Single Responsibility**
   - A class should focus on a single theme
   - A method should only focus on a single task

2. **Open/Close**
   - Open for extension but closed to modification
   - When a class is published, it must not be modified.
   - The only way to add or modify functionality is to create a subclass.

3. **Liskov Substitution Principle**
   - One class may be substituted for another, as long as the contract specified by the methods is fulfilled

4. **Interface segregation principle**
   - Each interface we create must be responsible for one and only one responsibility
   - A save interface must only perform a save operation.

5. **Dependency inversion principle**
   - The calling code must not manage dependencies. Instead, the caller should use a service to obtain these dependencies

2.3.PASS MADE RUGS

- **P**erformance
- **A**vailability
- **S**ecurity
- **S**calability

- **M**aintainability
- **A**ccessibility
- **D**eployability
- **E**xtensibility

- **R**esponsiveness
- **U**sability
- **G**lobalization
- **S**ex Appeal

2.4. STRIDE

- **S**poofing – Pretending to be someone else
- **T**ampering -
- **R**epudiation – Being anonymous
- **I**nformation Disclosure
- **D**enial of Services
- **E**levation of Privileges

3. **DESIGN PATTERNS**

Design patterns are things that software developers use all the time without thinking. Many design patterns are built into our modern programming languages. As a result, the use of design patterns is automatic.
This was different in the stone age of computers. Then all that was available was procedural programming. Then, it took conscious effort since it was all new unexplored territory.

Unfortunately, the recruiter will assume you know nothing and ask questions based on that assumption.

It is not enough to live and breathe the principles. You will need to spit out definitions in a way an only graduate student can.

3.1. Creational

- Factory
- Abstract Factory
- Builder
- Prototype
- Singleton

3.2. Structural

- Adaptor
- Bridge
- Composite
- Decorator
- Façade – Interface
- Flyweight
- Proxy

3.3. Behavioral

- Chain of Responsibility
- Command
- Iterator

- Mediator
- Momento
- Observer
- State
- Strategy
- Template Method
- Visitor

Here is a useful site for Design Patterns: https://refactoring.guru/design-patterns

4. DATA STRUCTURES & ALGORITHMS

Data structures are used to store the data a program needs to operate. Each data structure has advantages and disadvantages to their use.

Data structures are an essential study area on your quest for the job offer.

The following is a list of basic structures and interview questions. Keep in mind that when solving problems with data collections, linear solutions are always best.

As a reminder, going through a collection once for each element is a linear operation (n). The time it takes to go through the collection is linearly proportional to the number of elements in the collection.

Retrieving a number from an array is a constant operation (c). It always takes the same amount of time, regardless of the size of the collection.

Retrieving data from a hash table is a log operation (Log(n)).

The worst possible operation is the exponential operation (n^x). A classic example is the bubble sort algorithm. For each element of the collection, we go through the same collection at least once.

Example:
Let's say we have ten elements in a collection. It takes 1 second to process an element.

For a const operation (c), we need 1 second to solve the problem. For a linear operation (n), it could take a maximum of 10 seconds. For a bubble sort (n^2), we could take up to 10*10 or 100 seconds.

4.1. Strings and String Builder

String manipulation is a popular area in interviews. Here are a few examples:

- Reverse words in String.

ANS:

```
public void Rev(char[] str)
{
    if (str == null || str.Length <= 1) return;

    bool isWord = false;
    int wordStart = 0;

    for (int i = 0; i < str.Length; i++)
    {
        if (char.IsWhiteSpace(str[i]))
        {
            if (isWord)
            {
                RevWords(str, wordStart, i - 1);
                isWord = false;
            }
        }
        else
        {
            if (!isWord)
            {
                isWord = true;
                wordStart = i;
            }
        }
    }

    if (isWord)
    {
        RevWords(str, wordStart, str.Length - 1);
    }

    RevWords(str, 0, str.Length - 1);
}

private void RevWords(char[] str, int start, int end)
{
    while (start < end)
    {
        Var temp = str[start];
        str[start] = str[end];
        str[end] = temp;

        start++;
        end--;
    }
}
```

- Write a function to check if a given string is a palindrome. Ignore white spaces. Ex. “Able was I ere I saw Elba”

ANS:

Compare the first and last letters of a string. If they match, compare second and second last. You are done when you meet in the middle.

```
bool IsPal(String str)
{
    if (str == null) throw new ArgumentNullException();

    int s = 0;
    int e = str.Length - 1;

    while (s < e)
    {
        if (char.IsWhiteSpace(str[s]))
        {
            s++;
        }
        else if (char.IsWhiteSpace(str[e]))
        {
            e--;
        }
        else if (char.ToUpper(str[s]) == char.ToUpper(str[e]))
        {
            s++;
            e--;
        }
        else
        {
            return false;
        }
    }

    return true;
}
```

- Given a string s, find character c and return its position.

ANS:

```
int strchr(string s, char c)
{
    if (s == null) throw new ArgumentNullException();

    for (int i = 0; i < s.Length; i++)
    {
        if (s[i] == c) return i;
    }

    return -1;
}
```

4.2. Static Arrays

Static arrays are the standard arrays found implemented in most programming languages. It uses a fixed amount of space to store a fixed number of data elements.

For example:
string [] myStrings = new string {“String1”, “String2”, “String3”, “String4”};

You should be able to solve these problems in linear time.

- Bubble Sort an array of integers. (Included for completeness)

ANS:

```
void BubbleSort(int[] arr)
{
        if (arr == null || arr.Length <= 1) return;

        for (int i = 0; i < arr.Length - 1; i++)
        {
                for (int j = i + 1; j < arr.Length; j++)
                {
                        if (arr[i] > arr[j])
                        {
                                Var temp = str[i];
                                str[i] = str[j];
                                str[j] = temp;
                        }
                }
        }
}
```

- Reverse an array of integers

ANS:

```
public void RevInt(int[] arr)
{
    if (arr == null || arr.Length <= 1) return;

    int start = 0;
    int end = arr.Length - 1;

    while (start < end)
    {
        Var temp = str[start];
        str[start] = str[end];
        str[end] = temp;

        start++;
        end--;
    }
}
```

- Shuffle Cards

ANS:

```
public struct Card
{
    // Implimentation not relevant
}

void Shuffle(Card[] cards)
{
    if (cards == null || cards.Length <= 1) return;

    Random r = new Random();

    for (int i = 0; i < cards.Length; i++)
    {
        int j = r.Next(cards.Length);

        Card temp = cards[i];
        cards[i] = cards[j];
        cards[j] = temp;
    }
}
```

- Find shortest string in array of strings

ANS:

```
string ShortestString(string[] arr)
{
    if (arr == null) throw new ArgumentNullException();
    if (arr.Length == 0) throw new ArgumentException();

    int shortestLength = arr[0].Length;
    int shortestIndex = 0;

    for (int i = 1; i < arr.Length; i++)
    {
        if (arr[i].Length < shortestLength)
        {
            shortestLength = arr[i].Length;
            shortestIndex = i;
        }
    }

    return arr[shortestIndex];
}
```

- Given an array t[100] which contains all the numbers between 1..99. Return the duplicated value.

ANS:

```
int ReturnDup(int[] arr)
{
    // Assume array is correct.

    long sum = 0;
    foreach (int i in arr)
    {
        sum += i;
    }

    // N*(N + 1)/2
    long theoreticalMax = 99 * (99 + 1) / 2;

    long res = sum - theoreticalMax;
    if (res > 100) throw new ApplicationException();

    return (int)(sum - theoreticalMax);
}
```

- Find index of value in an array of sorted integers. Given value, return index or -1 if not found.

ANS:

```
public int BinSearch(int[] arr, int val)
{
    if (arr == null) throw new ArgumentNullException();
    if (arr.Length == 0) throw new ArgumentException();
    if (val < arr[0] || arr[arr.Length - 1] < val) return -1;

    int left = 0;
    int right = arr.Length - 1;
    int center = right / 2;

    while (true)
    {
        if (arr[center] == val) return center;
        if (left == right) return -1;

        if (arr[left] > arr[right]) throw new ArgumentException();

        if (val < arr[center])
        {
            right = center - 1;
        }
        else
        {
            left = center + 1;
        }

        center = (right - left) / 2 + left;
    }
}
```

- Remove duplicates from sorted array of integers.

ANS:

```
int[] RemoveDuplicates(int[] arr)
{
    if (arr == null || arr.Length == 0) return new int[0];

    List<int> newArr = new List<int>();
    newArr.Add(arr[0]);

    int ins = 0;
    for (int copy = 1; copy < arr.Length; copy++)
    {
        if (newArr[ins] != arr[copy])
        {
            ins++;
            newArr.Add(arr[copy]);
        }
    }

    return newArr.ToArray();
}
```

- Given an array containing both positive and negative integers, find the sub-array with the largest sum. What if you don't want negative numbers in the returned sub-array?

ANS:

```
int[] LargestSubArray(int[] arr)
{
    if (arr == null || arr.Length == 0) return new int[0];

    // Store current max sub-array
    int maxSum = arr[0];
    int maxStart = 0;
    int maxLength = 1;

    // Store working sub-array
    int currentSum = arr[0];
    int currentStart = 0;
    int currentLength = 1;

    for (int i = 1; i < arr.Length; i++)
    {
        // For only positive numbers,
        // Change to (currentSum <= 0 || Arr[i] <= 0)
        if (currentSum <= 0)
        {
            currentSum = arr[i];
            currentStart = i;
            currentLength = 1;
        }
        else
        {
            currentSum += arr[i];
            currentLength ++;
        }

        if (currentSum > maxSum)
        {
            maxSum = currentSum;
            maxStart = currentStart;
            maxLength = currentLength;
        }
    }

    int[] sub = new int[maxLength];
    for (int i = 0; i < maxLength; i++)
    {
        sub[i] = arr[maxStart + i];
    }

    return sub;
}
```

- Given an array, find if array contains two values that add to a given sum.
  bool TwoValuesThatSumExists(int [] arr, int sum)

ANS:

```
public bool TwoValuesThatSumExists(int[] arr, int sum)
{
    if (arr == null || arr.Length < 2)
        return false;

    var hash = new HashSet<int>();

    // To prevent the case where the first number
    // equals the sum, we add the first and then skip it.
    hash.Add(arr[0]);

    for (int i = 1; i < arr.Length; i++)
    {
        if (hash.Contains(sum - arr[i]))
        {
            return true;
        }
        else
        {
            hash.Add(arr[i]);
        }
    }

    return false;
}
```

- Combine 2 Sorted arrays
  - You have two arrays of unequal length.
  - Numbers in arrays are in ascending order.
  - The total number of non-zero numbers in both arrays equal size of the individual arrays.
  - Return an array with all the non-zero numbers in ascending order.

ANS:

```
public static int[] foo(int[] a1, int[] a2)
{
    if (a1 == null || a2 == null) throw new
ArgumentNullException();
    if (a1.Length != a2.Length) throw new ArgumentException();

    // The total number of non-zero numbers in both arrays equal
    // size of the individual arrays.
    int len = a1.Length;
    int[] res = new int[len];
    int resIndex = 0;

    int a1Index = 0;
    int a2Index = 0;

    while (resIndex < len)
    {
        if (a1Index < len && a1[a1Index] == 0)
        {
            a1Index++;
        }
        else if (a2Index < len && a2[a2Index] == 0)
        {
            a2Index++;
        }
        else
        {
            if (a1Index == a2Index && a1Index == len)
            {
                // not enough numbers
                throw new ApplicationException();
            }
            else if (a1Index == len)
            {
                // Add fron a1
                res[resIndex++] = a2[a2Index++];
            }
            else if (a2Index == len)
            {
                // Add from a2
                res[resIndex++] = a1[a1Index++];
```

```
            }
            else if (a1[a1Index] <= a2[a2Index])
            {
                res[resIndex++] = a1[a1Index++];
            }
            else
            {
                res[resIndex++] = a2[a2Index++];
            }
        }
    }

    return res;
}
```

- Using binary search, find smallest int in array that has been sorted and rotated an unknown number of times.
  Implement the following function:
  int FindSortedArrayRotation(int [] arr).

  - Arr is an array of unique integers that has been sorted in ascending order, then rotated by an unknown amount X, where 0 <= X <= (arrayLength - 1).
  - An array rotation by amount X moves every element array[i] to array[(i + X) % arrayLength].
  - Consider performance, memory utilization and code clarity and elegance of the solution when implementing the function.

ANS:

```
static int FindSortedArrayRotation(int[] arr)
{
    if (arr == null) throw new ArgumentNullException();
    if (arr.Length == 0) throw new ArgumentException();

    int left = 0;
    int right = arr.Length - 1;
    int center = right / 2;

    if (arr[left] <= arr[right]) return 0;

    while (true)
    {
        if (left + 1 == right)
        {
            if (arr[left] < arr[right])
            {
                return left;
            }
            else
            {
                return right;
```

```
            }
        }
        else if (arr[left] > arr[center])
        {
            right = center;
        }
        else
        {
            left = center;
        }

        center = (right - left) / 2 + left;
    }
}
```

- Given an array, find the one number that is repeated an odd number of times.

ANS:

```
public int GetOddCount(int[] arr)
{
    if (arr == null) throw new ArgumentNullException();
    if (arr.Length == 0) throw new ArgumentException();

    Dictionary<int, int> dict = new Dictionary<int, int>();

    foreach (int val in arr)
    {
        if (dict.ContainsKey(val))
        {
            dict[val] = dict[val] + 1;
        }
        else
        {
            dict[val] = 1;
        }
    }

    int numOddsFound = 0;
    int oddValue = 0;

    // Find value that is repeated an odd number of times
    foreach (int val in dict.Keys)
    {
        // Check if value is odd
        if (dict[val] % 2 == 1)
        {
            oddValue = val;
            numOddsFound++;
        }
    }

    if (numOddsFound == 0) throw new ApplicationException("No numbers found
repeated an odd number of times.");
    if (numOddsFound != 1) throw new ApplicationException("More than one
number found repeated an odd number of times.");

    return oddValue;
```

```
}
```

- Using the following function signature, write a C# function that prints out every combination of indices using Console.WriteLine() whose values add up to a specified sum, n. Values of 0 should be ignored.
  - public void PrintSumCombinations(List<int> numbers, int n);

  - It's okay to use additional private functions to implement the public function
  - Be sure to print out the indices of numbers and not the values at those indices
  - Don't worry too much about memory or CPU optimization; focus on correctness

To help clarify the problem, calling the function with the following input:

List<int> numbers = new List<int> { 1, 1, 2, 2, 4 };
PrintSumCombinations(numbers, 4);

Should result in the following console output (the ordering of the different lines isn't important and may vary by implementation):

0 1 2 (i.e. numbers[0] + numbers[1] + numbers[2] = 1 + 1 + 2 = 4)
0 1 3
2 3
4

ANS:

```
/// <summary>
/// Print all possible combinations of indexes that add to the number n.
/// Processing time increases exponentially with number of elements,
/// since we are doing a binary combination of all possible values
/// </summary>
/// <remarks>works for negative numbers as well</remarks>
public static void PrintSumCombinations(List<int> numbers, int n)
{
    if (numbers == null)
        throw new ArgumentNullException("numbers can't be null.");

    // Used to speed up processing
    var pairs = new List<Pair>();

    // Time needed to run increases exponentially with list size.
    // Reduce the workload as much as possible
    for (int i = 0; i < numbers.Count; i++)
    {
        // print elements with n
        if (numbers[i] == n)
        {
            // Only time when we print a single index
```

```
            System.Console.WriteLine(i);
        }
        else if (numbers[i] == 0)
        {
            // Ignore 0
            continue;
        }
        //else if (numbers[i] > n)
        //{
        //    // Add if negative numbers are forbidden
        //    continue;
        //}
        else
        {
            pairs.Add(new Pair() { index = i, value = numbers[i] });
        }
    }

    // Using brute force, we check all possible combinations of values
    var mask = new byte[pairs.Count + 1];
    IncrementBinaryArray(mask);

    while (mask[0] != 1)
    {
        // Using mask, find all possible combinations that add to n
        int temp = 0;
        for (int i = 0; i < pairs.Count; i++)
        {
            if (mask[i + 1] == 1)
                temp += pairs[i].value;
        }

        // Print only values that add to n
        if (temp == n)
        {
            for (int i = 0; i < pairs.Count; i++)
            {
                if (mask[i + 1] == 1)
                {
                    System.Console.Write(pairs [i].index + " ");
                }
            }

            System.Console.WriteLine();
        }
    }
}

/// <summary>
/// Increment a binary representation of the array we are examining
/// </summary>
private static void IncrementBinaryArray(byte[] arr)
{
    bool carry = true;
    for (int i = (arr.Length - 1); i >= 0; i--)
    {
        if (carry)
        {
            if (arr[i] == 0)
            {
```

```
                arr[i] = 1;
                carry = false;
            }
            else
            {
                arr[i] = 0;
                carry = true;
            }
        }
    }
 }

public struct Pair
{
    public int index;
    public int value;
}
```

4.3. Bit Arrays

We can use bit arrays when the possible values are finite. The advantage of bit arrays is that we can access the elements in constant time c. The disadvantage is that is consumes memory.

- Given an array of size N in which every number is between 1 and N, determine if there are any duplicates in it.

ANS:

```
bool HasDuplicates(int[] arr)
{
    if (arr == null) throw new ArgumentNullException();
    if (arr.Length <= 1) return false;

    BitArray ba = new BitArray(arr.Length);

    for (int i = 0; i < arr.Length; i++)
    {
        if (ba[arr[i] - 1]) return true;

        ba[arr[i] - 1] = true;
    }

    return false;
}
```

4.4. Hash Tables

Hash tables are a popular structure for solving problems associated with collections, since data can be retrieved in log time.

Hash tables work by using a hash key. This key is like the signature of the data that is being sorted.

- Given an array of unique integers, find all pairs of integers that add up to input number.

ANS:

```
public struct Pair
{
    public int Num1;
    public int Num2;
}

public List<Pair> GetPairs(int[] arr, int num)
{
    if (arr == null) return null;

    Hashtable ht = new Hashtable();
    List<Pair> res = new List<Pair>();

    for (int i = 0; i < arr.Length; i++)
    {
        int valueWeNeed = num - arr[i];

        if (ht.ContainsKey(valueWeNeed))
        {
            res.Add(new Pair{Num1 = arr[i], Num2 = valueWeNeed});
        }
        else
        {
            ht[arr[i]] = 1;
        }
    }

    return res;
}
```

- Given an arbitrary string, return a string with no duplicate characters.

ANS:

```
String RemDup(String s)
{
    if (s == null) throw new ArgumentNullException();

    // A bit array could have also been used.
    Hashtable ht = new Hashtable();

    StringBuilder sb = new StringBuilder();

    foreach (char c in s)
```

```
    {
        if (!ht.ContainsKey(c))
        {
            sb.Append(c);
            ht[c] = 1;
        }
    }

    return sb.ToString();
}
```

- Remove duplicates from unsorted array of integers.

ANS:

```
int[] RemDup(int[] arr)
{
    if (arr == null) throw new ArgumentNullException();
    if (arr.Length == 0) return new int[0];

    Hashtable ht = new Hashtable();

    for (int i = 0; i < arr.Length; i++)
    {
        ht[arr[i]] = 1;
    }

    int[] res = new int[ht.Count];
    ht.Keys.CopyTo(res, 0);

    return res;
}
```

- Find first element in array that is duplicated

ANS:

```
int FindDup(int[] arr)
{
    if (arr == null) throw new ArgumentNullException();
    if (arr.Length <= 1) return -1;

    Hashtable ht = new Hashtable();

    for (int i = 0; i < arr.Length; i++)
    {
        // Find duplicate
        if (ht.ContainsKey(arr[i]))
        {
            return (int)ht[arr[i]];
        }
        else
        {
            ht[arr[i]] = i;
        }
    }

    return -1;
}
```

- Find first element in array that is not duplicated

ANS:

```
int FindNonDup(int[] arr)
{
    if (arr == null) throw new ArgumentNullException();
    if (arr.Length == 0) throw new ArgumentException();

    Hashtable ht = new Hashtable();

    for (int i = 0; i < arr.Length; i++)
    {
        // Find duplicate
        if (ht.ContainsKey(arr[i]))
        {
            ht[arr[i]] = 2;
        }
        else
        {
            ht[arr[i]] = 1;
        }
    }

    for (int i = 0; i < arr.Length; i++)
    {
        if (((int)ht[arr[i]]) == 1)
        {
            return i;
        }
    }

    return -1;
}
```

4.5. Stacks

Stacks are like stacks of pates. You can only add and remove plates from the top.

- How would you implement a queue only with stacks?

ANS:

Have 2 stacks: addStack, delStack
To add, move contents to addStack, then add
To delete, move contents to delStack, then delete

4.6. Queues

Queues are like lines of people going to the bank. You enter at the tail of the line and the people at the head of the line get processed.

➢

ANS:

4.7. Linked Lists

Linked lists are one of the most popular areas in interviews. Knowing them is essential.

All answers here assume this definition of a node:

```
public class Node<T>
{
    public Node<T> Next;
    public T Value;
}
```

➢ Insert element into linked list.

ANS:

```
void Insert<T>(ref Node<T> head, Node<T> newNode) where T :
IComparable<T>
{
    if (head == null)
    {
        head = newNode;
        return;
    }

    if (newNode.Value.CompareTo(head.Value) <= 0)
    {
        newNode.Next = head;
        head = newNode;
        return;
    }

    Node<T> curr = head;
    while (curr.Next != null)
    {
        if (newNode.Value.CompareTo(curr.Next.Value) <= 0)
        {
            newNode.Next = curr.Next;
            curr.Next = newNode;
            return;
        }

        curr = curr.Next;
    }
```

```
    curr.Next = newNode;
}
```

- Delete element from linked list.

ANS:

```
void Delete<T>(ref Node<T> head, T val) where T : IComparable<T>
{
    if (head == null) return;

    if (head.Value.CompareTo(val) == 0)
    {
        head = head.Next;
        return;
    }

    Node<T> c = head;
    while (c.Next != null)
    {
        if (c.Next.Value.CompareTo(val) == 0)
        {
            c.Next = c.Next.Next;
            return;
        }

        c = c.Next;
    }
}
```

- Find middle element.

ANS:

```
Node<T> Middle<T>(Node<T> head) where T : IComparable<T>
{
    if (head == null) return null;
    if (head.Next == null) return head;

    Node<T> slow = head;
    Node<T> fast = head.Next;

    while (true)
    {
        if (fast == slow) throw new
         ApplicationException("Circular linked Lists are not allowed");

        if (fast == null) return slow;
        if (fast.Next == null) return slow.Next;

        fast = fast.Next.Next;
        slow = slow.Next;
    }
}
```

- Delete alternate nodes in doubly linked list.

ANS:

```
void DeleteAltNodes<T>(Node<T> head) where T : IComparable<T>
{
    Node<T> c = head;
    while (c != null && c.Next != null)
    {
        c.Next.Next.Previous = c;
        c.Next = c.Next.Next;
        c = c.Next;
    }

    c.Next = null;
}
```

- Find M^{th} to last element in linked list.

ANS:

```
Node<T> MthNode<T>(Node<T> head, int mth) where T : IComparable<T>
{
    if (head == null) throw new ArgumentNullException();
    if (mth < 1) throw new ArgumentException("mth must be 1 or greater");

    Node<T> curr = head;
    int i = 1;

    while (i < mth && curr.Next != null)
    {
        curr = curr.Next;
        i++;
    }
    if (i < mth) throw new ApplicationException("mth node doesn't exist");

    Node<T> res = head;
    while (curr.Next != null)
    {
        curr = curr.Next;
        res = res.Next;
    }

    return res;
}
```

- Remove duplicates in sorted list.

ANS:

```
void RemoveDuplicates<T>(Node<T> head) where T : IComparable<T>
{
    if (head == null) return;

    Node<T> curr = head;
    while (curr.Next != null)
    {
        if (curr.Value.CompareTo(curr.Next.Value) == 0)
        {
            curr.Next = curr.Next.Next;
        }
```

```
            curr = curr.Next;
        }

        curr.Next = null;
    }
```

- How would you design an algorithm to find if there is a cycle in a singly linked list?

ANS:

```
public static bool IsCircle<T>(Node<T> head)
{
    if (head == null || head.Next == null) return false;

    Node<T> slow = head;
    Node<T> fast = head.Next;

    while (true)
    {
        if (slow == fast || slow.Next == fast) return true;

        if (fast == null || fast.Next == null) return false;

        fast = fast.Next.Next;
        slow = slow.Next;
    }
}
```

- Merge two sorted lists (ascending).

ANS:

```
Node<T> Merge<T>(Node<T> l1, Node<T> l2) where T : IComparable<T>
{
    if (l1 == null) return l2;
    if (l2 == null) return l1;

    Node<T> c1 = l1;
    Node<T> c2 = l2;
    Node<T> c;
    Node<T> res;

    if (c1.Value.CompareTo(c2.Value) < 0)
    {
        c = c1;
        c1 = c1.Next;
    }
    else
    {
        c = c2;
        c2 = c2.Next;
    }
    res = c;

    while (true)
    {
        if (c1 == null)
```

```
        {
            c.Next = c2;
            return res;
        }
        else if (c2 == null)
        {
            c.Next = c1;
            return res;
        }
        else if (c1.Value.CompareTo(c2.Value) < 0)
        {
            c.Next = c1;
            c = c.Next;
            c1 = c1.Next;
        }
        else
        {
            c.Next = c2;
            c = c.Next;
            c2 = c2.Next;
        }
    }
}
```

- Reverse linked list.

ANS:

```
void Reverse<T>(ref Node<T> head)
{
    if (head == null) return;

    Node<T> next = head.Next;
    head.Next = null;

    while (next != null)
    {
        Node<T> prev = head;
        head = next;
        next = next.Next;
        head.Next = prev;
    }
}
```

- Given 2 lists L1 and L2, write a function intersect which would return the element common in both L1 and L2. Do this for both sorted and unsorted arrays. What if sizeof(L1) << sizeof(L2).

ANS:

- Given a reference to the middle of a linked list, remove the node the reference is pointing to.

ANS:

4.8. Binary Search Trees

Binary search trees form a major research area. You can insert/delete elements in Log(n) time. It also makes good use of memory, since you only allocate memory when needed.

All answers here assume this definition of a node:

```
public class Node<T>
{
    public Node<T> Left;
    public Node<T> Right;
    public T Value;
}
```

- Insert node into binary search tree

ANS:

```
public void Insert<T>(ref Node<T> root, Node<T> newNode) where T :
IComparable<T>
{
    if (root == null)
    {
        root = newNode;
        return;
    }

    Node<T> curr = root;

    while (true)
    {
        if (newNode.Value.CompareTo(curr.Value) == 0)
        {
            return;
        }
        else if (newNode.Value.CompareTo(curr.Value) < 0)
        {
            if (curr.Left == null)
            {
                curr.Left = newNode;
                return;
            }
            else
            {
                curr = curr.Left;
            }
        }
        else
        {
```

```
            if (curr.Right == null)
            {
                curr.Right = newNode;
                return;
            }
            else
            {
                curr = curr.Right;
            }
        }
    }
}
```

- Delete node from binary search tree.

ANS: (Relies on previous answer)

```
void Delete<T>(ref Node<T> root, T val) where T: IComparable<T>
{
    if (root == null) return;

    if (root.Value.CompareTo(val)==0)
    {
        Node<T> temp = root.Right;
        root = root.Left;
        Insert(ref root, temp);
    }

    Node<T> curr = root;

    while (true)
    {
        if (curr.Value.CompareTo(val) > 0 )
        {
            if (curr.Left == null)
            {
                return;
            }
            else if (curr.Left.Value.CompareTo(val) == 0)
            {
                Node<T> temp = curr.Left.Right;
                curr.Left = curr.Left.Left;
                Insert(ref curr.Left, temp);
            }
            else
            {
                curr = curr.Left;
            }
        }
        else
        {
            if (curr.Right == null)
            {
                return;
            }
            if (curr.Right.Value.CompareTo(val)==0)
            {
                Node<T> temp = curr.Right.Right;
                curr.Right = curr.Right.Left;
```

```
                Insert(ref curr.Right, temp);
            }
            else
            {
                curr = curr.Right;
            }
        }
    }
}
```

- Find Lowest Common Ancestor, given:
  Node LCA(Node root, int left, int right);

ANS:

```
Node<T> LCA<T>(Node<T> root, T left, T right) where T : IComparable<T>
{
    if (root == null) throw new ArgumentNullException();
    if (right.CompareTo(left) < 0) throw new ArgumentException();

    Node<T> curr = root;

    while (true)
    {
        if (left.CompareTo(curr.Value) <= 0 &&
            curr.Value.CompareTo(right) <= 0)
        {
            return curr;
        }
        else if (left.CompareTo(curr.Value) < 0 &&
                 right.CompareTo(curr.Value) <= 0)
        {
            if (curr.Left == null)
            {
                return curr;
            }
            else
            {
                curr = curr.Left;
            }
        }
        else
        {
            if (curr.Right == null)
            {
                return curr;
            }
            else
            {
                curr = curr.Right;
            }
        }
    }
}
```

- Write a function to find the depth of a binary tree.

ANS:

```
int Dept<T>(Node<T> root)
{
    if (root == null) return 0;

    int left = Dept(root.Left);
    int right = Dept(root.Right);

    return 1 + (left > right ? left : right);
}
```

- Print elements in-order (Left, Center, and Right).

ANS:

```
void Print<T>(Node<T> root)
{
    if (root == null) return;

    Print(root.Left);
    Console.WriteLine(root.Value);
    Print(root.Right);
}
```

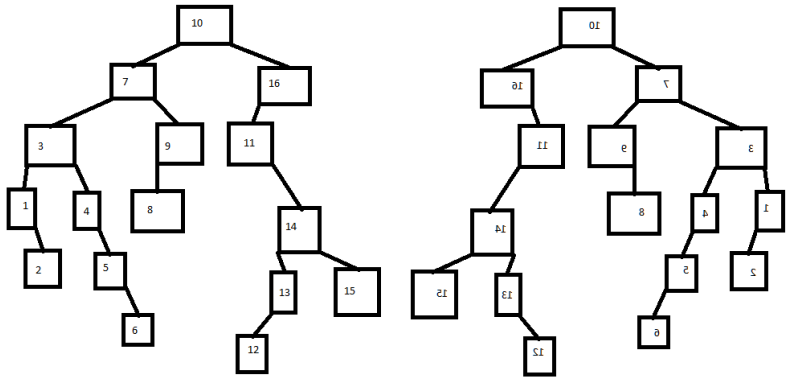
- Convert BST into a linked list.

ANS:

```
Node<T> Convert<T>(Node<T> root)
{
    if (root == null)
    {
        return null;
    }

    Node<T> head = new Node<T>();
    Node<T> curr = head;
    Unravel(ref curr, root);

    // Get rid of first (dummy) node
    curr = head;
    head = head.Left;

    curr.Left = null;
    curr.Right = null;
    head.Left = null;

    return head;
}

private void Unravel<T>(ref Node<T> head, Node<T> root)
{
    if (root.Left != null)
    {
        Unravel(ref head, root.Left);
    }

    head.Right = root;
    root.Left = head;
```

```
    head = root;

    if (root.Right != null)
    {
        Unravel(ref head, root.Right);
    }
}
```

⌦ Given two Binary trees, find if they are mirror images of one another.



These trees are reflections of one another.

ANS:

```
public bool IsSymetrical<T>(Node<T> left, Node<T> right) where T :
IComparable<T>
{
        if (left == null && right == null)
        {
                return true;
        }
        else if (left == null || right == null)
        {
                return false;
        }
        else
        {
                return (left.Value.CompareTo(right.Value) == 0) &&
                        IsSymetrical(left.Left, right.Right) &&
                        IsSymetrical(left.Right, right.Left);
        }
}
```

5. GENERAL TECHNICAL QUESTIONS

This area deals with technical questions that don't fit into any area. They don't rely on collections of data but test general programming aptitude.

5.1. General Questions

- Given the time in hours and minutes, find the angle between the hour hand and the minute hand.

ANS:

```
public int GetAngleBetweenClockHands(int hours, int minutes)
{
    int hourAngle = 360 / 12;
    int minuteAngle = 360 / 60;

    return hourAngle * hours - minuteAngle * minutes;
}
```

- Parse integer from string.

ANS:

```
int Parse(string s)
{
    if (s == null) throw new ArgumentNullException();

    bool isNeg = false;
    bool numStart = false;
    bool numEnd = false;
    int res = 0;

    foreach (char c in s)
    {
        if (c == '-')
        {
            if (numStart) throw new FormatException();

            isNeg = true;
            numStart = true;
        }
        else if (c == ' ')
        {
            if (numStart) numEnd = true;
        }
        else if ('0' <= c && c <= '9')
        {
            if (numEnd) throw new FormatException();

            res *= 10;
```

```
            res += (int)(c - '0');
            numStart = true;
        }
        else
        {
            throw new FormatException();
        }
    }

    if (isNeg) res *= -1;

    return res;
}
```

- Find factorial of n. (0, 1, 2, 3) => (1, 1, 2, 6)

ANS:

```
int NthFactorial(int n)
{
    if (n < 0) throw new ArgumentException();

    if (n == 0) return 1;

    int res = 1;

    for (int i = 1; i <= n; i++)
    {
        res *= i;
    }

    return res;
}
```

- Find nth Fibonacci number.
  Ex: (0, 1, 2, 3, 4, 5, 6) => (0, 1, 1, 2, 3, 5, 8)

ANS:

```
int NthFibonacci(int nth)
{
    if (nth < 0) throw new ArgumentNullException();

    int n1 = 0;
    int n2 = 1;

    for (int i = 0; i < nth; i++)
    {
        int temp = n1 + n2;
        n1 = n2;
        n2 = temp;
    }

    return n2;
}
```

- Write a function that takes as its arguments 3 integers > 0.
  If the product of the integers is even, then return the one with the lowest value, else show 0.

ANS:

```
int LowestOrZero(int i1, int i2, int i3)
{
    if (i1 <= 0 || i2 <= 0 || i3 <= 0) throw new ArgumentException();

    // If all ints are odd, return 0;

    if ((i1 & 1) == 1 && (i2 & 1) == 1 && (i3 & 1) == 1) return 0;

    int t = i1 < i2 ? i1 : i2;
    return t < i3 ? t : i3;
}
```

- Print integer one digit at a time, without storing any intermediate results.

ANS:

```
void PrintNum(int num)
{
    if (num == 0)
    {
        Console.WriteLine("0");
        return;
    }

    if (num < 0)
    {
        num *= -1;
        Console.WriteLine("-");
    }

    int mask = GetMask(num);

    while (num > 0)
    {
        int units = num / mask;
        Console.Write(units);
        num -= units * mask;
        mask /= 10;
    }
}

int GetMask(int num)
{
    if (num == 0) return 0;

    float res = 0.1F;
    while (num > 0)
    {
        res *= 10;
        num /= 10;
    }
```

```
    return (int)res;
}
```

- Draw 1/8 of a circle.

ANS:

```
void DrawEightCircle(double rad)
{
    double x = 0;
    double y = rad;

    while (x <= y)
    {
        y = Math.Sqrt((rad * rad - x * x) + 0.5);
        SetPixel(x, y);
        x++;
    }
}
```

5.2. Floating Point Operations

- Why should we use decimal over float variable?

ANS:

A floating point variable (float) suffers from rounding errors.
Decimal does not.

Test 1:
var aa = 1.2 - 1.0;
Console.WriteLine(aa);
Ans: 0.2

Test 2:
var bb = (decimal)(1.2 - 1.0)
Console.WriteLine(bb);
Ans: 0.2

NOTE: To see the error, you will need to use the debugger. Look in the locals window of the Visual Studio.

In locals window:
aa = 0.19999999999999996
bb = 0.2M

NOTE: In Visual Studio, to use the C# interactive console (View | Other Windows | C# Interactive)

NOTE: To run in Visual Studo Code, you will need the 'Polyglot Notebooks' extension.

5.3. Bit Operations

- Find number of set bits in an integer.

ANS:

```
int BitsSet(int num)
{
    uint ui = (uint)num;

    uint res = 0;

    while (ui > 0)
    {
        res += ui & 1;
        ui >>= 1;
    }

    return (int)res;
}
```

- Give a one-line expression to test if a number is a power of 2.

ANS:

```
public bool IsPowerOf2(int num)
{
    return ((num > 1) && ((num & (num - 1)) == 0));
}
```

- There are two integers, 'a' and 'b'. Swap the contents without using a temp variable.

ANS:

```
int a=1, b=2;

a ^= b;
b ^= a;
a ^= b;
```

- Multiply num by 7 without using multiplication (*) operator.

ANS:

```
int DivBy7(int num)
{
        return num << 3 - num;
}
```

- Print integer in base 2.

ANS:

```
void Print(int num)
{
    if (num == 0) Console.WriteLine("0");
```

```
    uint uNum = (uint)num;

    uint temp = uNum;
    int mask = 0;
    while (temp > 0)
    {
        mask++;
        temp >>= 1;
    }

    while (mask > 0)
    {
        mask--;
        Console.Write(uNum & (1 << mask));
        uNum >>= 1;
    }
}
```

5.4. Recursion

- Find nth factorial. (0, 1, 2, 3) => (1, 1, 2, 6)

ANS:

```
int NthFact(int n)
{
    if (n <= 0) throw new ArgumentException();

    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * NthFact(n - 1);
    }
}
```

- Find nth Fibonacci number.
  Ex: (0, 1, 2, 3, 4, 5, 6) => (0, 1, 1, 2, 3, 5, 8)

ANS:

```
int Fib(int n)
{
    if (n < 0) throw new ArgumentNullException();

    if (n == 0) return 0;
    if (n == 1) return 1;

    return Fib(n - 1) + Fib(n - 2);
}
```

- Print numbers from 1 to 100 and then 100 to 1 without any for or while loops.

ANS:

```
public void Print100()
{
    Print(1);
}

private void Print(int n)
{
    Console.Write(n);

    if (n < 100) Print(n + 1);

    Console.Write(n);
}
```

- Count the number of zeros in an array of integers without loops.

ANS:

```
public int CountZeros(int[] arr)
{
    if (arr == null) throw new ArgumentNullException();
    if (arr.Length == 0) throw new ArgumentException();

    return Count(arr, 0);
}

private int Count(int[] arr, int i)
{
    int n = 0;
    if (arr[i] == 0)
    {
        n = 1;
    }

    if (i < arr.Length)
    {
        n += Count(arr, i++);
    }

    return n;
}
```

- Print linked list in reverse order.

ANS:

```
public class Node<T>
{
    public Node<T> Next;
    public T Value;
}

void Print<T>(Node<T> root)
{
    if (root == null) return;

    Print(root.Next);
    Console.Write(root.Value);
```

```
}
```

- Find Greatest Common divisor of two numbers.

ANS:

```
public static uint gcd(uint n1, uint n2)
{
    if (n2 == 0) return n1;
    return gcd(n2, n1 % n2);
}
```

- Sum the digits in a number.
  Ex: If x is 2345, return 9 [2 + 3 + 4 + 5]

ANS:

```
int Count(int num)
{
        if (num < 0) num *= -1;
        if (num == 0) return 0;
        return Count(num / 10) + num % 10;
}
```

- Reverse a singly linked list recursively.

ANS:

```
void RevList<T>(ref Node<T> head)
{
    Rev(ref head, null);
}

private void Rev<T>(ref Node<T> head, Node<T> prev)
{
    if (head == null) return;

    Node<T> tmp = head.Next;
    head.Next = prev;

    prev = head;
    head = tmp;

    Rev(ref head, prev);
}
```

5.5. C++ Questions

- Find if computer is big-endian or little-endian (C++).

ANS:

```
// Returns 1 if machine is little-endian, 0 otherwise.
int IsLittleEndian()
```

```
{
    union
    {
        int theInterger;
        char singleByte;
    } endianTest;

    endianTest.theInterger = 1;

    return endianTest.singleByte;
}
```

- Change a 4-byte number from big-endian to little-endian.

ANS:

```
public int BigToSmallEndian(int num)
{
    return (num & (255 << 24)) >> 8 +
           (num & (255 << 16)) << 8 +
           (num & (255 << 8)) >> 8 +
           (num & (255)) << 8;
}
```

5.6. Advanced Problems

- Print all permutations in a string.

ANS:

```
void Print(string s)
{
    if (s == null) return;

    for (int i = 0; i < s.Length; i++)
    {
        Console.Write(s[i]);
        Print(GetSub(s, i));
        Console.WriteLine();
    }
}

private string GetSub(string s, int i)
{
    if (s == null || s.Length <= 1) return null;
    if (i < 0 || i >= s.Length) throw new ArgumentException();

    StringBuilder sb = new StringBuilder();

    for (int j = 0; j < i; j++)
    {
        sb.Append(s[j]);
    }

    for (int j = i + 1; j < s.Length; j++)
    {
```

```
            sb.Append(s[j]);
        }

        return sb.ToString();
    }

    private int Count(int num)
    {
        if (num < 0) num *= -1;
        if (num == 0) return 0;
        return Count(num / 10) + num % 10;
    }
```

6. TESTING/DEBUGGING

Testing is fundamental to the software development process. In the

6.1. General Questions

- What does the following code output?

Unit tests are used to define how a class is supposed to behave in various scenarios. By fully defining expectations, we can swap components with reduced fear of having unintended consequences. (Polymorphism, Inversion of Control, etc.)

6.2. C++ Questions

- What does the following code output?

Code:	
`` void foo() { uint a = 6; int b = -20; string res = (a + b > 6) ? ">6" : "<=6"; Console.WriteLine(res); } ```	
ANS (C++):	In C++, `res = ">6"` **Reason:** Expressions involving signed and unsigned types have all operands promoted to unsigned types. Thus -20 becomes a very large positive integer and the expression evaluates to greater than 6.
ANS (C#):	In C#, `res = "<=6"` **Reason:** both uint and int are converted to long